SHõWA: A Self-Healing Framework for Web-Based Applications

JOÃO PAULO MAGALHÃES, CIICESI, ESTGF-Polytechnic Institute of Porto
LUIS MOURA SILVA, CISUC, University of Coimbra

The complexity of systems is considered an obstacle to the progress of the IT industry. Autonomic computing is presented as the alternative to cope with the growing complexity. It is a holistic approach, in which the systems are able to configure, heal, optimize, and protect by themselves. Web-based applications are an example of systems where the complexity is high. The number of components, their interoperability, and workload variations are factors that may lead to performance failures or unavailability scenarios. The occurrence of these scenarios affects the revenue and reputation of businesses that rely on these types of applications.

In this article, we present a self-healing framework for Web-based applications (*SHõWA*). *SHõWA* is composed by several modules, which monitor the application, analyze the data to detect and pinpoint anomalies, and execute recovery actions autonomously. The monitoring is done by a small aspect-oriented programming agent. This agent does not require changes to the application source code and includes adaptive and selective algorithms to regulate the level of monitoring. The anomalies are detected and pinpointed by means of statistical correlation. The data analysis detects changes in the server response time and analyzes if those changes are correlated with the workload or are due to a performance anomaly. In the presence of performance anomalies, the data analysis pinpoints the anomaly. Upon the pinpointing of anomalies, *SHõWA* executes a recovery procedure. We also present a study about the detection and localization of anomalies, the accuracy of the data analysis, and the performance impact induced by *SHõWA*. Two benchmarking applications, exercised through dynamic workloads, and different types of anomaly were considered in the study. The results reveal that (1) the capacity of *SHõWA* to detect and pinpoint anomalies while the number of end users affected is low; (2) *SHõWA* was able to detect anomalies without raising any false alarm; and (3) *SHõWA* does not induce a significant performance overhead (throughput was affected in less than 1%, and the response time delay was no more than 2 milliseconds).

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Reliability

Additional Key Words and Phrases: Self-healing, autonomic computing, Web applications, fault tolerance, performance

ACM Reference Format:
João Paulo Magalhães and Luis Moura Silva. 2015. *SHõWA*: A self-healing framework for Web-based applications. ACM Trans. Autonom. Adapt. Syst. 10, 1, Article 4 (March 2015), 28 pages.
DOI: http://dx.doi.org/10.1145/2700325

1. INTRODUCTION

Web applications are a class of business-critical applications. *Internet Retailer* magazine reveals that in 2012, the turnover from online sales grew 19% in Latin

Authors' addresses: J. P. Magalhães, CIICESI, ESTGF-Polytechnic Institute of Porto; email: jpm@estgf.ipp.pt; L. M. Silva, CISUC, University of Coimbra; email: luis@dei.uc.pt.

DOI: http://dx.doi.org/10.1145/2700325

America, reaching $15.42 billion; 16% in the United States, reaching $225.54 billion; 16% in Europe, reaching $302.20 billion; and 32% in Asia, reaching $256.50 billion. Unfortunately, and despite all efforts that have been made, these applications are still affected by performance and availability issues. These issues have a direct impact on a company's revenue, customer satisfaction, and employees productivity.

Bojan Simic, the president and principal analyst at TRAC Research, reports that the average revenue loss for 1 hour of Web site downtime is $21,000, and the average revenue loss of Web site slowdown is estimated in $4,100 per hour [Simic 2010]. His study states that Web site slowdowns may occur 10 times more frequently than outages. Another report, provided by the Aberdeen Group [2010], states that a delay of just 1 second in page-load time can represent a loss of $2.5 million in sales per year for a site that typically earns $100,000 a day. The impact of user dissatisfaction is hard to estimate. A study from PhoCusWright and Akamai [Rheem 2010] states that 8 out of 10 people will not return to a site after a disappointing experienc, and of these, 3 people will go on to tell others about their experience. Sean Power [2010] found that 37% to 49% of users who experience performance issues when completing a transaction will either abandon the site or switch to a competitor. Of these, 77% will share their experience with others.

Analysis with regard to the cause of failures shows that failures primarily are motivated by three type of problems: hardware, software, and human error. Although hardware and network problems have been controlled over the past years, software failures and human error still face a big challenge. A report about the cause of failures in Web applications [Pertet and Narasimhan 2005] indicates that 40% of failures are motivated by operator error, and the other 40% are due to application failures. The reasons behind these failures are largely related to the complexity of the systems, inadequate testing, or poor understanding of system dependencies.

Aware of these challenges, Paul Horn from IBM launched the concept of autonomic computing (AC) in 2001 [Ganek and Corbi 2003]. The ultimate aim of AC is to automate the management of computing systems to address its increasingly complexity. With the adoption of "self" attributes, such as self-configuring, self-healing, self-optimizing, and self-protecting, systems will be able to adapt automatically to the environment and be capable of reacting to abnormal scenarios. Likewise, the involvement of IT staff will be smaller, thus reducing the risk of human error.

In this article, we present a self-healing framework for Web-based applications (*SHõWA*). The framework aims to provide Web-based applications with the ability to detect performance anomalies at runtime and trigger automatic recovery actions to mitigate their impact. *SHõWA* fits the *fail-stutter* fault model. This model takes into account performance-faulty scenarios, in which a component provides unexpectedly low performance but continues to function correctly with regard to its output. *SHõWA* makes use of aspect-oriented programming (AOP) to collect system, application server, and application-level data from the managed resource at runtime. The data is prepared and submitted to statistical correlation analysis (Spearman's rank correlation) to distinguish performance anomalies from workload variations and pinpoint performance anomalies at different levels (system, application server, application, local or remote changes). It also includes recovery procedures that are automatically executed upon the detection and localization of a performance anomaly. We also present the results of an experimental study about the ability of *SHõWA* to detect and pinpoint performance anomalies, the performance impact induced by the framework, and the accuracy of the data analysis provided by *SHõWA*. The study considers different types of anomalies, injected into two Web benchmarking applications and exercised through dynamic workloads.

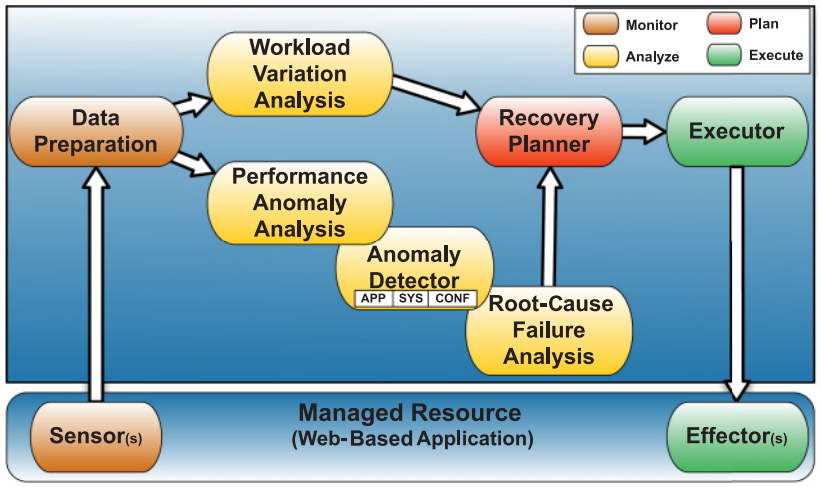


Fig. 1. Building blocks of the *SHõWA* framework.

The rest of the article is organized as follows. Section 2 describes the *SHõWA* framework. The experimental setup and the results about the detection and pinpointing of anomalies are presented in Section 3. In Section 4, we present the results about the accuracy of. The performance impact induced by the framework is presented in Section 5. Related work is presented in Section 6, and Section 7 concludes the article.

2. *SHõWA* FRAMEWORK

The *SHõWA* framework is illustrated in Figure 1. It resembles an autonomic element: the *managed resource* corresponds to the Web-based application and its execution environment, and the *autonomic manager* iterates over four processes: Monitor, Analyze, Plan, and Execute.

The *Sensor* module collects data from different system and application server parameters and measures the execution time of user transactions and application calls. The data is sent to a remote database to be prepared by the *Data Preparation* module. The data preparation involves the data aggregation in time intervals; the creation of a unique key at the end of each interval, which identifies the mix of transactions in a given interval; and the creation of an association key to identify the list of calls belonging to the user transaction. The *Workload Variation Analysis* and *Performance Anomaly Analysis* modules carry out statistical analysis to detect response time variations and to verify if the response time variations are due to workload changes or are a symptom of a performance anomaly. If there is a variation in the response time, motivated by a performance anomaly, then the *Anomaly Detector* module carries out statistical analysis to detect if there is any change in the system or application server parameters correlated with the performance anomaly; the *Root-cause Failure Analysis* module carries out statistical analysis to analyze the response time of the application calls to check if there are changes in the application, or in the invocation of remote services, that are correlated with the performance anomaly.

Upon the detection and localization of performance anomalies, the recovery phase follows. The recovery phase involves the *Recovery Planner* module, which is responsible for selecting a recovery procedure, and the *Executor* module, which dispatches the recovery actions to be applied on the *managed resource* by the *Effector* module.

2.1. SHõWA Fault Models

The type of failures to be addressed is one of the most important considerations made. In the context of dependable systems, *SHõWA* can be seen as a fault tolerance and reliability system. In the area of fault tolerance, there are three fault models: fail-stop, Byzantine, and fail-stutter. The fail-stop fault model [Schneider 1990] is characterized as a model in which upon the occurrence of a failure, all operations are stopped. The Byzantine fault model [Lamport et al. 1982] considers that a process affected by a fault can continue its execution producing incorrect results. The fail-stutter fault model [Arpaci-Dusseau and Arpaci-Dusseau 2001] takes into account that the components of a system sometimes fail and sometimes perform erratically (e.g., low performance) but are not reflected in the final results. These unexpected behaviors are defined as performance faults.

Whereas the Byzantine fault model is considered general and difficult to apply, the fail-stop fault model is considered too simple and inadequate to represent the behavior of modern systems. The fail-stutter model extends the fail-stop model by taking into account performance-faulty scenarios. *SHõWA* targets the detection and recovery of performance anomalies to mitigate the negative impacts caused by a performance slowdown. In this context, the fail-stutter model is the most appropriate fault model for *SHõWA*.

2.2. Monitoring: *SHõWA Sensor* Module

System-level monitoring and end-to-end monitoring systems are commonly adopted to detect problems in Web-based applications. System-level monitoring observes the system parameters periodically and triggers alerts when the status of the parameters does not match the expected state. End-to-end monitoring is used to check the status of the service as it is experienced by the end users. The end-to-end monitoring systems typically execute, in a periodic manner, one transaction or a set of transactions to check the availability of the service, the occurrence of errors, and the response time. Another monitoring system that is gaining expression within Web applications is application-level monitoring. This type of monitoring complements the previous ones. It provides specific data on the state of the application. For example, in addition to indicating if a transaction is slow, these systems can indicate whether the slowness is due to the state of the resources in the system (e.g., CPU load). The *SHõWA* framework performs application-level monitoring at runtime. It gathers the execution time of the application transactions and collects data from the system to detect performance anomalies and pinpoint if the anomaly is motivated by a system change, a change in the application server, or an application change.

Collect application-level data in production Web-based applications presents some challenges. The access to the application source code usually is limited. Even if it is possible, current enterprise applications are made up of multiple and heterogenous components that cross the enterprise boundaries, making the understanding of the source code a complex and daunting task.

The *Sensor* module included in *SHõWA* enables application-level monitoring. It is a small program implemented according to the (AOP paradigm [Kiczales et al. 1997]. This program is installed with the application server and allows collection of information about the applications that run on the server, as well as the state of the system parameters and application server parameters. To activate the *Sensor*, we just need to put the program in a directory of the application server so that when the application server is started, the monitoring program is automatically loaded. Once loaded, the *Sensor* module intercepts the running application every time a match between the method/call signature and the AOP pointcuts defined in the *Sensor* module source code

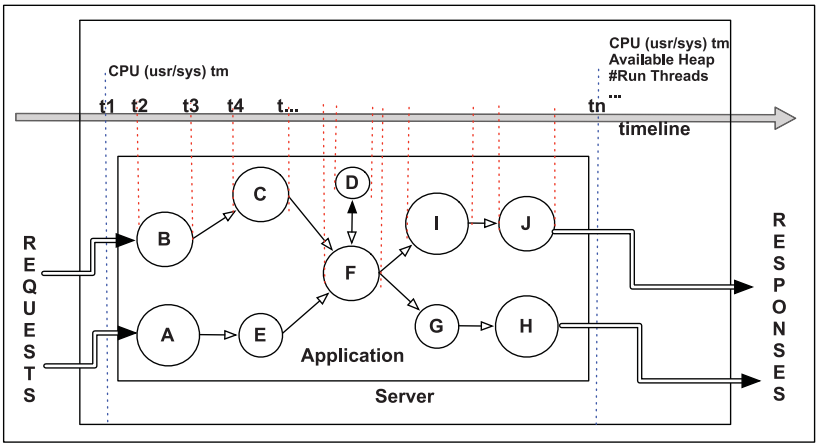


Fig. 2. Application-level monitoring: user-transaction interception and measurement in a user-transaction mode (t_1 and t_n) and profiling mode (t_2 to t_{n-1}).

Table I. Example of Parameters Collected by the *Sensor* Module

Parameter	T	Description
User transactions	$t_n - t_1$	Duration per user transaction (ms)
CPU (trans.)	$t_n - t_1$	CPU time per user transaction
Memory	t_n	Amount of available memory
Threads	t_n	Number of running and created threads
Transaction	t_1	Signature/Transaction name
File Descriptors	t_n	Number of open and max number of files allowed
Classes	t_n	Number of running and created classes
CPU (OS)	t_n	Percentage of CPU USR, SYS, and WIO usage
Net traffic	t_n	Amount of bytes received and sent
Disk traffic	t_n	Amount of bytes written and read
Profiling	$t_2 ... t_{n-1}$	Duration per user-transaction call (ms)

is verified. The type of instrumentation, provided by the AOP, is known as bytecode instrumentation. It is advantageous because it separates the monitoring code from the application code, thereby allowing an application to be monitored without the need for manually changing its source code. With this separation, multiple applications can be monitored using the same monitoring code.

The *SHõWA Sensor* module collects data with two different levels of granularity: *user-transaction*–level monitoring and *profiling*-level monitoring. As illustrated in Figure 2, at the user-transaction level, it intercepts and measures the server response time of user transactions and gathers different system and application server parameters (e.g., CPU load, JVM heap memory, number of open files, number of running threads). At the *profiling* level, it intercepts and records the execution time of the calls involved in the user-transaction call path. The data collected at the user-transaction level is used to detect slow user transactions and identify if there is a system or application server parameter change that is associated with the performance anomaly. The application profiling data is used to pinpoint the calls/components associated with a performance anomaly.

In Table I, we present the list of parameters that are collected by default by the *Sensor* module. This list can be extended as necessary.

To minimize the performance impact induced by application-level profiling, we considered the use of adaptive and selective monitoring algorithms. These algorithms dynamically adapt the behavior of the *Sensor* module, reducing or increasing the

frequency of application profiling. Next we present the algorithms currently implemented in the framework.

2.2.1. Adaptive Monitoring. The application *profiling* performed by the *Sensor* module measures the processing time of all application calls. In practice, this corresponds with taking a timestamp before and after each line of code executed. With such low-level data, we can verify if there are application calls associated with a response time slowdown. Collection of low-level data increases the risk of seriously affecting the performance of the system under monitoring. To mitigate this problem, the *Sensor* module adopts an adaptive behavior. It includes adaptive and selective algorithms to self-adjust the frequency to which the *Sensor* module profiles the application calls, as well as dynamically selects the list of calls that should be intercepted.

Currently, it is possible to choose between one of three algorithms: linear, exponential, and polynomial.

Linear Adaptation Algorithm. In the linear adaptation algorithm, the sampling frequency adjustment is proportional to the decrease or increase verified in the correlation degree computed by the *Performance Anomaly Analysis* module.

The linear adaptation algorithm is shown in Algorithm 1. We use the notation M to denote the maximum sampling frequency and m to denote the minimal sampling frequency. The average correlation degree per user transaction, $\bar{x}$, is derived from the historical correlation degree, and r refers to the last correlation degree computed. The α corresponds to a correlation degree decrease, which is representative of a symptom of anomaly.

ALGORITHM 1: *SHõWA* user-transaction profiling: linear adaptation of the sampling frequency

```
1: procedure LinearAdaptation(M,m,x̄,r,α)
2:     K ← floor(M − ((x̄ − r) ∗ (M/α)))
3:     if (K > M) then
4:         K ← M
5:     end if
6:     if (K < m) then
7:         K ← m
8:     end if
9:     Return K
10: end procedure
```

The $\bar{x} - r$ corresponds to the decrease observed in the correlation degree. The M/α corresponds to the adjustment to make to M, per unit of correlation degree above or below $\bar{x}$. The algorithm produces the value K, which is the amount of adjustment to be applied. It takes a value between M and m and corresponds to the sampling frequency used by the *Sensor* module. The higher the value of K, the lower the sampling frequency. The lower the value of K, the higher the sampling frequency.

Exponential adaptation algorithm. The exponential adaptation algorithm improves on the linear adaption algorithm by considering the adjustment of the sampling frequency K using an exponentiation factor (Equation (1)) that depends on M and α:

$$M = e^{F\alpha}. \tag{1}$$

On solving the Equation (1), we get the exponentiation factor F (Equation (2)):

$$F = \frac{\ln M}{\alpha}. \tag{2}$$

By using an exponentiation factor for adjusting the frequency of user-transaction profiling, we have a moderate frequency adjustment when the variation in the correlation degree is small and a faster adjustment in the frequency when the change observed in the correlation degree is close to the defined correlation degree threshold (α).

The exponential adjustment of the frequency of user-transaction profiling is given in Algorithm 2.

ALGORITHM 2: *SHōWA* user-transaction profiling: Exponential adaptation of the sampling frequency

```
1: procedure ExponentialAdaptation(M,m,x̄,r,α)
2:     Q ← x̄ − r;
3:     if (Q ≤ 0) then
4:         Q ← 1
5:     end if
6:     if (Q > M) then
7:         Q ← M
8:     end if
9:     K ← floor(e^(F∗Q)) − M          ▷ F is determined by Equation (2)
10:    if (K < m) then
11:        K ← m
12:    end if
13:    Return K
14: end procedure
```

With Algorithm 2, the sampling frequency remains large when r deviates only a few degrees from $\bar{x}$. This way, the amount of data collected is reduced and the performance impact induced is minimized. For significant deviations, the sampling frequency increases, accelerating the application-level profiling frequency and providing more data, to support the data analysis process used to pinpoint the anomaly.

Polynomial adaptation algorithm. The polynomial adaptation algorithm fits between the linear and exponential algorithms. It is composed with different exponential functions and allows one to keep the sampling K large until a given amount/percentage (identified by β) of correlation degree decrease is observed. After that point, the sampling frequency K can be adapted more frequently to provide enough data for the data analysis process used to pinpoint anomalies.

For example, considering a polynomial function of order three ($f(x) = ax^3 + bx^2 + cx + d$) and a β value in percentage, the polynomial adaptation function can be achieved by solving the system of equations presented in Equation (3):

$$\begin{cases} M = a + b + c + d \\ M\beta = (\alpha\beta)^3 a + (\alpha\beta)^2 b + \alpha\beta c + d \\ M\beta^2 = (\alpha 2\beta)^3 a + (\alpha 2\beta)^2 b + \alpha 2\beta c + d \\ m = \alpha^3 a + \alpha^2 b + \alpha c + d. \end{cases} \tag{3}$$

Considering a system of linear equations with n equations and n variables, the independent terms a, b, c, and d of Equation (3) can be solved using Cramer's rule. For each variable, the denominator is the determinant of the matrix of coefficients, whereas the numerator is the determinant of a matrix in which one column has been replaced by the vector of constant terms. The set of equations used to solve the terms

a, b, c, and d is presented in Equation (4):

$$a = \frac{\begin{vmatrix} M & 1 & 1 & 1 \\ M\beta & (\alpha\beta)^2 & \alpha\beta & 1 \\ M\beta^2 & (\alpha 2\beta)^2 & \alpha 2\beta & 1 \\ m & \alpha^2 & \alpha & 1 \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 1 \\ (\alpha\beta)^3 & (\alpha\beta)^2 & \alpha\beta & 1 \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & \alpha 2\beta & 1 \\ \alpha^3 & \alpha^2 & \alpha & 1 \end{vmatrix}} \quad b = \frac{\begin{vmatrix} 1 & M & 1 & 1 \\ (\alpha\beta)^3 & M\beta & \alpha\beta & 1 \\ (\alpha 2\beta)^3 & M\beta^2 & \alpha 2\beta & 1 \\ \alpha^3 & m & \alpha & 1 \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 1 \\ (\alpha\beta)^3 & (\alpha\beta)^2 & \alpha\beta & 1 \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & \alpha 2\beta & 1 \\ \alpha^3 & \alpha^2 & \alpha & 1 \end{vmatrix}} \tag{4}$$

$$c = \frac{\begin{vmatrix} 1 & 1 & M & 1 \\ (\alpha\beta)^3 & (\alpha\beta)^2 & M\beta & 1 \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & M\beta^2 & 1 \\ \alpha^3 & \alpha^2 & m & 1 \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 1 \\ (\alpha\beta)^3 & (\alpha\beta)^2 & \alpha\beta & 1 \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & \alpha 2\beta & 1 \\ \alpha^3 & \alpha^2 & \alpha & 1 \end{vmatrix}} \quad d = \frac{\begin{vmatrix} 1 & 1 & 1 & M \\ (\alpha\beta)^3 & (\alpha\beta)^2 & \alpha\beta & M\beta \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & \alpha 2\beta & M\beta^2 \\ \alpha^3 & \alpha^2 & \alpha & m \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & 1 \\ (\alpha\beta)^3 & (\alpha\beta)^2 & \alpha\beta & 1 \\ (\alpha 2\beta)^3 & (\alpha 2\beta)^2 & \alpha 2\beta & 1 \\ \alpha^3 & \alpha^2 & \alpha & 1 \end{vmatrix}}$$

The corresponding polynomial function is then used within the polynomial adaptation algorithm to define the sampling frequency K. The value of K is given in Algorithm 3.

ALGORITHM 3: *SHõWA* user-transaction profiling: polynomial adaptation of the sampling frequency

```
1: procedure PolynomialAdaptation(M,m,x̄,r,α.β)
2:     Q ← x̄ − r;
3:     if (Q ≤ 0) then
4:         Q ← 1
5:     end if
6:     if (Q > M) then
7:         Q ← M
8:     end if
9:     K ← floor(aQ^3 + bQ^2 + cQ + d)        ▷ a,b,c, and d: determined according to Equation (3)
10:    if (m > 1 and K < m) then
11:        K ← m
12:    end if
13:    Return K
14: end procedure
```

By using Algorithm 3, the sampling frequency is adapted at three different rates. If the correlation degree decrease is below a given threshold (β), then the adaptation is very small—that is, the sampling frequency will remain large. If the correlation degree degree is close to the α value, then the adaptation will occur more quickly, allowing collection of more low-level data to support the data analysis used to pinpoint anomalies. Between one and another, the adaptation is moderate.

Selective monitoring. Whereas the adaptation algorithm redefines the sampling frequency as the correlation degree between user-transaction response time and the

number of concurrent user-transaction changes, the selective monitoring technique adjusts the list of calls/components profiled by the *Sensor* module.

A user transaction can hold a considerable number of calls with different contributions to the total response time. Rather than intercepting all of the calls, the *Sensor* module takes advantage of the AOP pointcut to decide, at runtime, if a call should be intercepted or not. To do this, the pointcut consults a data structure. If the call exists in the data structure, then the body of the pointcut is executed and the call response time is measured; if the call does not exist in the data structure, then the body of the pointcut is not executed.

The data structure contains all calls/components that have a contribution over **X**% to the total user-transaction response time. From $K * R$ times (with K being the sampling frequency and R a refresh factor), all calls are intercepted, and the calls with a processing time higher than **X**% of the user-transaction response time are inserted in the data structure. The data structure is also updated in a reflexive manner (i.e., when the correlation degree decreases). A call is removed from the data structure when its response time is less than **X**% of the user-transaction response time.

With this approach, the number of calls to be intercepted is reduced. By reducing the number of calls, the performance impact induced by the *Sensor* module will be lower.

2.3. Monitoring: *SHõWA Data Preparation* Module

The *Data Preparation* module is responsible for preparing the data collected by the *Sensor* module. This module includes two data preparation functions: the data interval and the workload mix key. The data interval function is used to determine the time interval to which the collected data belongs. Its operation is very simple and essential to the data analysis process. The lower limit of the first interval (L_1) corresponds to the timestamp of the first transaction. The upper limit is achieved by summing S seconds to the lower limit ($U_1 = L_1 + S$). Starting from the first, all intervals are sequential: the lower limit of the t^{th} interval is given by $L_t = U_{t-1} + 1$, and the upper limit is given by $U_t = L_t + S$. The transaction response time and the state of the system and application server parameters collected by the *Sensor* module are assigned to an interval whenever its timestamp is between the interval limits, which is determined by the data interval function.

The workload mix key function is used to characterize the mix of user transactions processed in an interval. This characterization is important because the resource demands may vary according to the mix of user transactions processed. Doing this identification makes it possible to compare the response time between the intervals under analysis. The definition of the workload mix key, for a given interval, is given by Equation (5):

$$key_t = hash(\%tm_1 \ldots \%tm_n)_t, \tag{5}$$

where

$$\%tm_i = \frac{\sum_t UserTransaction_i}{\sum_t UserTransactions}.$$

Equation (5) returns a *key* value based on the percentage of user transactions processed in a given interval (t). To reduce the key value space, each $\%tm_i$ is rounded to the nearest percentage that is multiple of 5% (e.g., $0.17 \geq 0.15$ and $0.18 \geq 0.20$).

2.4. Analyze: *SHõWA* Detecting and Pinpointing Anomalies

The data analysis to detect and pinpoint performance anomalies and workload anomaly scenarios is performed by the *Workload Variation Analysis*, *Performance Anomaly Analysis*, *Anomaly Detector*, and *Root-Cause Failure Analysis* modules.

The data analysis considers the state of the system parameters, the application container parameters, and the application response time to detect performance anomalies and identify whether the cause is related to workload changes, system or application server changes, or application changes. By detecting application-level issues that occur in the server, it becomes possible to prevent the occurrence of performance failures before they affect a wide range of end users.

The data analysis is based on Spearman's rank correlation coefficient, commonly represented by the Greek letter ρ (rho) [Zar 1972]. The correlation coefficient is given by Equation (6), and it expresses how two variables ($\boldsymbol{X}$ and $\boldsymbol{Y}$) are associated. It works by calculating Pearson's correlation coefficient on the ranked values of the data ($\boldsymbol{X}$ and $\boldsymbol{Y}$). Ranking is obtained by assigning a rank, from low to high, to the values of $\boldsymbol{X}$ and $\boldsymbol{Y}$, respectively. Spearman's correlation does not require the data to be normality distributed (it is a nonparametric statistic). The determination of ρ and the subsequent significance testing require the data to be measured in intervals and the variables to be monotonically related. These assumptions are both fulfilled: the data is aggregated in time intervals; the elements of $\boldsymbol{X}$ and $\boldsymbol{Y}$ form a pair.

$$\rho = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \tag{6}$$

From the analysis point of view, ρ can be interpreted as follows [Cohen 1988]: between $[-1.0, -0.5]$ or $[0.5, 1.0]$ there is a large correlation, between $[-0.49, -0.3]$ or $[0.3, 0.49]$ there is a medium correlation, between $[-0.29, -0.1]$ or $[0.1, 0.29]$ there is a small correlation, and between $[-0.09, 0.09]$ there is no correlation.

As in Kelly and Zhang [2006], the data analysis exploits typical properties of modern enterprise distributed applications:

—Workload consists of request-reply transactions.
—Transactions occur in a small number of types.
—Resource demands vary widely across but not within transaction types.
—Computational resources are adequately provisioned, so transaction times consist largely of service times, not queueing times.
—Crucial aspects of workload are statistically nonstationary (the frequency distributions of key workload characteristics (e.g., mix and load) vary dramatically over time).

Besides the preceding properties, in Web-based applications the following commonly apply:

—A single user transaction with the system is relatively short lived.
—The server response times vary widely across but not within user-transaction types.

In addition to the preceding assumptions, it is noted that for a correct functioning of the data analysis process, it is important to guarantee that an initial period of normal behavior exists—that is, a period when there are no anomalies affecting the performance of the application. This initial period, according to the experiments conducted, corresponds to the execution of 100 transactions.

Another aspect that may influence the interpretation of the results has to do with the performance stability delivered by the infrastructure where the application is deployed. By *performance stability*, we refer to the ability of a server or virtual machine to provide consistent performance over time. Whereas in a traditional cluster-based infrastructure the performance stability is easily achieved (identical physical resources are exclusively dedicated to a single application), in a virtualized/cloud infrastructure, regardless of the good level of isolation enforced by the hypervisor, the performance

stability is more difficult to guarantee due to the existence of multiple virtual machines sharing resources and with different workload types because of infrastructure operations such as virtual machine creation, migration, scaling, and destroy. Due to stability differences, a higher number of performance changes and detection by *SHõWA* in a Web application hosted in a virtualized/cloud environment are expected compared to a Web application hosted in a traditional cluster environment.

With regard to the impact that performance stability may have in the analysis produced, the following should be noted:

—Independent from the performance stability guarantees provided by the infrastructure, *SHõWA* will be able to detect the performance anomalies affecting the application—that is, any significant change in the response time that is not correlated with the number of requests will be detected through the data analysis process.
—The ability to pinpoint performance anomalies is highly influenced by the type of infrastructure. In a dedicated infrastructure, it is more simple to determine the cause of the anomaly and select the appropriate recovery procedure. In a virtualized/cloud infrastructure, it will be necessary to deepen the analysis to determine if a performance anomaly is correlated with events considered normal in terms of infrastructure operation (e.g., new virtual machine creation) or if there are performance anomalies that need to be addressed immediately (e.g., lack of resources in the physical host or virtual machine). In this context, *SHõWA* presents a great potential for cross checking between the application and infrastructure state, allowing the various actors (service provider and service consumer) to have a more complete view about the quality of the service delivered and to take the most appropriate recovery actions quickly and when necessary.

Performance Anomaly Analysis: detecting performance anomalies. To detect if there exists a performance anomaly affecting a Web-based application, the *Performance Anomaly Analysis* module primarily organizes the data. It defines a vector $\boldsymbol{X}$, which contains the sequence of the accumulated response server time per user transaction and a vector $\boldsymbol{Y}$ that holds contains the number of user transactions processed in the same interval. Then, using Spearman's rank correlation, it measures the degree of association between $\boldsymbol{X}$ and $\boldsymbol{Y}$. In terms of analysis, the result provided by Spearman's rank correlation is simple. Assuming an initial period of normal functioning, if ρ remains stable and high across the periods of analysis, then it means that the response time is associated with the application workload. If ρ decreases, then it means that one of the variables has increased or decreased while the other has remained stable or changed in an opposite direction. Considering the typical properties of modern distributed applications, presented earlier, this dissociation corresponds to a symptom of performance anomaly.

Despite the simplicity of the data analysis, there are some aspects that require a more detailed analysis. Assuming an increase in the accumulated response time accompanied by an increase in the number of transactions processed seems logical. However, in this scenario, the ρ may evince a strong relationship and therefore obfuscating scenarios where the response time is already affecting the end users. Another challenge is that from the analysis, it is not possible to know if the dissociation is due to a change in the response time or due to a change in the number of transactions. Likely, from the ρ degree variation, it is not possible to quantify the response time variations.

To address these limitations, the *Performance Anomaly Analysis* module proceeds as described in Algorithm 4. Considering p a new interval for analysis, n the number of previous intervals, $MART$ a maximum admissible response time increase, and K a workload mix key, this algorithm measures the difference between the ρ degree observed in a given interval and the maximum value of ρ observed previously. The

same is done with the response time. At the end, it returns the product between the variance of ρ and the variance of response time.

The value of F, returned by Algorithm 4, quantifies the impact of dissociation between the response time and the number of user transactions processed. A value of F is obtained for each user transaction.

ALGORITHM 4: Performance Anomaly Analysis: degree of dissociation between the response time and the number of user transactions processed

```
1: procedure PerformanceAnomalyAnalysis(dataset)
2:     for each UserTransaction do
3:         X[] ← Accumulated Response Time UserTransaction      per p and K
4:         Y[] ← Number of User-Transactions UserTransaction    per p and K
5:         ρ[p] ← Spearman Rank Correlation(X̄[], Ȳ[]))
6:         V1 ← Avg_RespTime[p] − Avg_RespTime      [p − n : (p − 1)/2]
7:         if (V1 > MART) then
8:             V1 ← MAXINT
9:         end if
10:        V2 ← ρ[p] − Max_ρ[p − n : (p − 1)/2]
11:        F[UserTransaction] ← √(V1 ∗ V2)²
12:    end for
13:    Return F
14: end procedure
```

The adoption of Spearman's rank correlation to identify the disassociation between the response time and the application workload presents several advantages. One of the key advantages comes from its mathematical properties. Spearman's rank correlation is invariant to separate changes in the scale of the variables. Therefore, $\boldsymbol{X}$ or $\boldsymbol{Y}$ can assume values of different magnitudes and do not affect the correlation coefficient. This allows use of the same type of analysis for all user transactions and thus detects when the response time of a transaction is no longer aligned with the number and mix of transactions processed. When compared to other methods of statistical correlation (Pearson and Kendall), we found advantages to adoption of Spearman's correlation. In contrast to Pearson correlation, Spearman's correlation is a nonparametric statistic, so it can be used when the data follows a distribution that is not known a priori. This argument is strengthened because the response time and the number of users typically follow a data distribution with long tails (non-Gaussian). Between Kendall correlation and Spearman's correlation, we have chosen Spearman's correlation because this method is more robust to the presence of outliers. Thus, occurrences of peaks in the response time or number of users do not affect the data analysis, reducing the occurrence of false alarms.

Proof of concept. In Table II, we present a simulation considering the value F returned by the *Performance Anomaly Analysis* module. The simulation considers different response time delays and different variations on the number of user transactions processed. For the simulation, we used 200 pairs of $\boldsymbol{X}$ and $\boldsymbol{Y}$. Five new pairs of data were added to the end of $\boldsymbol{X}$ and $\boldsymbol{Y}$, representing the variations under test. Each pair contains the data aggregated in time intervals of 5 seconds.

From Table II, it is clear that the $\boldsymbol{F}$ value is higher when the response time increases and the number of user transactions processed decreases. When the number of user transactions processed varies but the response time remains stable, the value $\boldsymbol{F}$ is close to zero. For a response time delay equal to or higher than 500 milliseconds, the

Table II. Determining the Threshold Value That Quantifies the Impact of a Performance Anomaly

Number of User Transaction Variations (%) → Response Time Delay (ms) ↓	1	0.5	0	–0.5	–1
+2000	18.902	41.665	89.349	335.178	1794.432
+500	4.518	10.120	22.045	83.972	454.622
+100	0.682	1.708	4.097	16.983	97.340
+50	0.202	0.656	1.854	8.610	52.679
+25	0.006	0.082	0.497	3.346	23.867
0	0.006	0.007	0.006	0.022	0.52

value of $\boldsymbol{F}$ increases independently from the variations observed in the number of user transactions processed. According to the analysis, a response time delay of just 50 milliseconds originates in a value of $\boldsymbol{F}$ higher than 10 only after 90 intervals, a deviation of 100 milliseconds originates in a value of $\boldsymbol{F}$ higher than 10 after 60 intervals, and a deviation of 500 milliseconds is detected immediately at the second interval.

The results presented in Table II are useful to help define a global threshold value that can be used to highlight the occurrence of a performance anomaly. For example, a value of $\boldsymbol{F}$ higher than 10 can be used to detect performance slowdowns that clearly deviate from the number of user transactions processed.

Workload Variation Analysis: detecting workload problems. A workload variation might occur due to a change in the transaction mix or a change in the load (number of transactions). In a Web-based application, both the mix and the number of transactions are highly variable parameters. The number of users varies over time, and the pages they visit are determined by their own interests. These features make the modeling of workload a challenging task.

The analysis provided by the *Performance Anomaly Analysis* module contemplates changes in the workload mix and load. In that module, the data is organized according to a key that corresponds to the mix of transactions. Per mix and number of transactions, the data analysis detects delays in response time and verifies if the changes result from variations in response time. One cause of such variation may be a workload problem (e.g., bursty workload, server queue issues). In this context, the *Workload Variation Analysis* module completes the data analysis. It measures the correlation between the observed response time and the number of requests waiting to be processed. To perform this analysis, the *Workload Variation Analysis* module defines $\boldsymbol{X}$ as the accumulated response time in a given time interval and $\boldsymbol{Y}$ as the total number of requests waiting in the application container queue in the same time interval.

The *Workload Variation Analysis* module makes use of Spearman's rank correlation coefficient $\boldsymbol{\rho}$. Assuming the existence of previous intervals already analyzed and that the service was performing under normal conditions, the $\boldsymbol{\rho}$ degree is expected to be stable and low across the time intervals. The fact that the $\boldsymbol{\rho}$ degree is low and stable means two things: (1) the response time has not suffered delays, and (2) there was no significant accumulation of requests in the application server queue waiting to be processed. If the $\boldsymbol{\rho}$ degree is medium or large, then it means that there are problems with the processing workload.

In the presence of performance anomalies that according to the analysis provided by the *Workload Variation Analysis* module are not associated with a workload problems, then the *Anomaly Detector* and *Root-Cause Failure Analysis* modules continue with the data analysis process. These modules search for system, application server, or application changes to pinpoint the cause behind the response time slowdown.

Anomaly detector: looking for system or application server changes. After detecting a performance anomaly, the *Anomaly Detector* module aims to identify if there is any system or application server change associated with the anomaly.

The *Anomaly Detector* module takes $\boldsymbol{X}$ as the total number of user transactions processed in an interval and $\boldsymbol{Y}$ as the accumulated value of the parameters collected by the *Sensor* module in the same interval. There is a vector $\boldsymbol{Y}$ for each parameter collected. The data is grouped using the workload mix key, and it is continuously analyzed as new time intervals become available. As in the previous analyses, the module computes Spearman's rank correlation coefficient between the variables.

For example, rather than stating how much a parameter $\boldsymbol{P}$ changes according to the number and mix of user transactions in a given time interval, the analysis provided by the *Anomaly Detector* module tells us if the value of a given parameter increases or decreases according to the number of requests processed. Considering how the data is prepared for the analysis, if the number of requests increases, then the accumulated value of each parameter should also increase. If the accumulated value of a given parameter increases and is not motivated by an increase in the number of requests, then the ρ degree will decrease, showing that a parameter (or set of parameters) is no longer aligned with the number of requests.

Given the sequence of data analysis presented until now, it is possible to determine if the application is facing a performance anomaly and to verify if the anomaly is associated with changes in the parameters analyzed by the *Anomaly Detector* module. Pinpointing the parameters associated with a performance anomaly is extremely important in deciding the most appropriate recovery strategy and in reducing the time to recover.

Root-Cause Failure Analysis: looking for application or remote service changes. To verify if a performance anomaly is associated with an application change or a remote service change, the variable X is defined as the frequency distribution of the transaction response time and Y assumes the response time frequency distribution of the calls belonging to the transaction call path. The correlation between the variables is determined using Spearman's rank correlation coefficient. The *Root-Cause Failure Analysis* module only analyzes the user transactions that have reported a performance anomaly in the previous analyses.

Due to the non-Gaussian nature of the data, we used Doane's formula (Equation (7)) to determine the number of data bins to discretize the variables $\boldsymbol{X}$ and $\boldsymbol{Y}$. To determine the number of bins, it is necessary to calculate the kurtosis of the distribution $\bar{a}$ (measure related to the shape "peakedness" of the data distribution) and provide the number of observations under analysis (n).

$$Nbins = 1 + \log_e n + \log_e \left(1 + \bar{a}\sqrt{\frac{n}{6}}\right) \tag{7}$$

The number of bins returned by Equation (7) is used to discretize the variables $\boldsymbol{X}$ and $\boldsymbol{Y}$. Whereas one of the variables indicates the frequency of the response time of a user transaction, the other contains the response time frequency for the calls belonging to the user transaction under analysis. From Spearman's rank correlation, it is expected that the association between the variables is stable unless there is one or more calls that the response time has changed together with a change in the response time of the user transaction. In this situation, the ρ degree increases, highlighting the calls potentially associated with the performance slowdown.

The identification of the calls associated with a delay in response time allows, for example, one to consult the change management database to see if the problem is related to any recent application change. It also allows one to find patterns in the methods that

are causing problems. For example, several calls about queries in the database can be linked to problems in the database tier, and calls regarding the interaction with Web services can highlight problems with a Web service.

2.5. Recovery: *SHõWA* Planning and Executing Recovery

The recovery service included in the *SHõWA* framework is provided by three modules. The *Recovery Planner* module contains the recovery procedures, ready to be activated when an anomaly is detected. This module is activated by the *Workload Variation Analysis* after it detects a problem affecting the application workload or by the *Anomaly Detector* and *Root-Cause Failure Analysis* modules after a performance anomaly is detected and pinpointed. In this module, there exists a set of actions that can be grouped to create a recovery procedure. These actions indicate, for example, what to do to inform a load balancer to stop sending requests to a given server or what command should be executed to stop the application server.

In the current implementation, the recovery process is procedure based and defined by a human operator. The operator identifies the recovery actions to be included in the recovery procedure and sets the rules in which the procedure is activated. The operator can define more than one recovery procedure for the same rule. To that end, it should identify the priority and the atomicity of the recovery actions. The priority corresponds to a numerical value that describes the expected effect after an action is performed. For example, if it is expected that a server restart provides better recovery results than a simple application restart, then it must have a higher priority number. The atomicity is a Boolean value that defines if a recovery action can be interrupted at any time or not. This value is important to prevent service inconsistencies motivated by the recovery process (e.g., forcing a restart while the application is being downgraded may lead to an inconsistent application version or compromise the application startup).

The recovery procedures remain under the *Recovery Planner* module, and they are selected according to the type of anomaly detected. Once selected, a recovery procedure is executed by the *Executor* module. The *Executor* module is responsible for the interaction with the *Effector* module, passing to it the sequence of recovery actions and controlling its execution. The communication between the nodes is made through a client–server program. The client sends the actions to be performed to the server and receives feedback about its implementation. The *Executor* module knows how many systems are being monitored and controls how many of these are in recovery. This control process is done to avoid service failures motivated by the execution of simultaneous recovery processes. This module also controls the execution of multiple recovery requests to a given node. By default, when a node is under recovery, it does not accept new recovery actions. Exceptions are allowed when two conditions are simultaneously met: (1) the new repair action has a higher recovery priority when compared to the recovery action in progress, and (2) the repair action in execution can be interrupted at any time (atomic bit is set to false). If a recovery procedure fails or it takes more than a given threshold to be executed, then a different recovery procedure can be taken.

An example of a recovery procedure, defined for a performance anomaly motivated by an application change, and considering an infrastructure provided with load balancing (LB) and high-availability (HA) services, is is presented in the chart.

The recovery process is thought to evolve to become more autonomous. The current implementation gathers data about the execution of the recovery actions: success of the recovery procedure, time taken by the recovery procedure, and the number of errors that have occurred during the recovery process. The main idea is to combine utility functions and machine learning algorithms to build a system able to learn with previous recovery actions and decide which recovery actions should be taken in future

Performance anomaly motivated by an application change: recovery procedure

```
1: Decrement the number of appserver instances available (synchronized)
2: if (number of instances available >= 1) then
3:     Deploy a previous version of Web App
4:     Remove the control file that is checked by the load-balancer
5:     Do a rolling downgrade to switch the Web App version
6:     Restore the control file that is checked by the load-balancer
7: end if
8: if (appserver instance under recovery is available) then
9:     Increment the number of appserver instances available
10:    Repeat the recovery process on the other appserver instances
11: else
12:    Send an alert to the console
13: end if
```

situations (e.g., perform the recovery procedure that is the fastest; perform the recovery procedure that cause fewer errors).

3. *SHõWA*: DETECTING AND PINPOINTING ANOMALIES—EXPERIMENTAL STUDY

In this section, we present an experimental study conducted to assess the ability of *SHõWA*: to detect and pinpoint different types of performance anomalies, analyze the performance impact induced by the framework, and evaluate the accuracy and precision of the data analysis. Specifically, we seek answers to the following questions:

—When there is a performance variation? Was it caused by a change in the workload, or is it a result of some internal anomaly?
—How does one distinguish between a performance anomaly and workload variations?
—How does one obtain useful information about the potential root cause?
  —Is the performance anomaly motivated by a resource contention in the server?
  —Is the performance anomaly motivated by an application server contention?
  —Is the performance anomaly associated with an application change?
  —Is the performance anomaly associated with a remote service change?
—How timely is the detection and pinpointing of anomalies?
—How accurate is the detection and pinpointing of anomalies?
—What is the performance impact induced by *SHõWA*?

The study considers two benchmark applications and different anomaly scenarios.

3.1. Benchmark Applications

For the experimental study, we have adopted two J2EE benchmarking applications: TPC-W [Smith 2001] and the Rice University Bidding System (RUBiS) [Cecchet et al. 2002].

The TPC-W benchmark simulates the activities of a retail store Web site. It defines 14 user transactions that are classified as either of browsing and ordering types. It uses a MySQL database as a storage backend and contains its own workload generator that emulates the behavior of end users according to three workload mixes: (1) browsing mix (95% of Web browsing and 5% of Web ordering), (2) ordering mix (50% of Web browsing and 50% of ordering), and (3) shopping mix (80% of browsing and 20% of ordering).

The RUBiS benchmark application models an auction site. Loosely modeled on eBay, it defines 26 user transactions. Among the most important user transactions are browsing items by category or region, bidding, buying or selling items, leaving comments about other users, and consulting one's own user page. Browsing items also includes consulting the bid history and the seller's information. RUBiS includes two workload

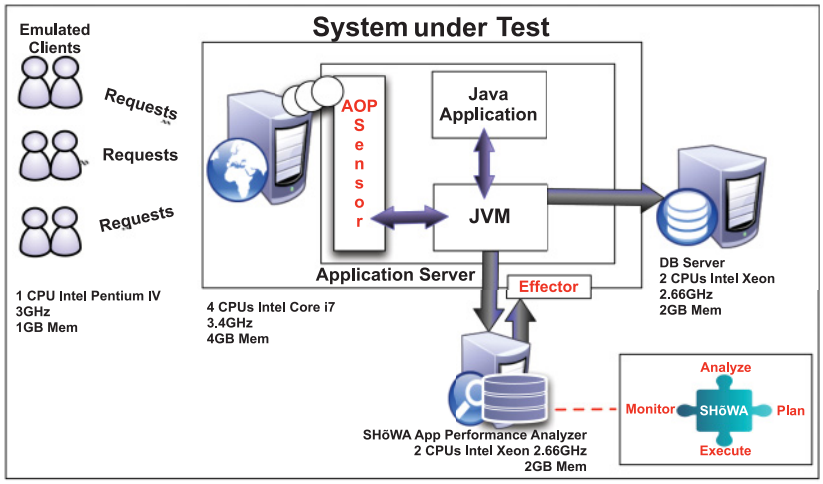


Fig. 3. Test environment.

mixes: a browsing mix made up of 100% of read-only interactions and a bidding mix that includes 15% of read-write interactions and 85% of read-only interactions.

3.2. Workloads

TPC-W contains its own workload generator that simulates the activities of a business-oriented transactional Web server. It applies the workload via emulated browsers (RBE) and allows the execution with different concurrent users, three different traffic mixes, different session length, user think times, and ramp-up/ramp-down periods. According to the TPC-W specification, the number of emulated browsers and workload mix is kept constant during the experiment. However, since real e-commerce applications are characterized by dynamic workloads, we created workloads that change the number of users and traffic mix during the execution. The workload duration is 8,400 seconds; while it is executing, the workload mix changes every 5 to 15 minutes and the number of emulated users varies between 30 and 500.

Like TPC-W, RUBiS includes a client–browser emulator. It defines a session as a sequence of interactions for the same customer. The client emulator opens a persistent HTTP connection to the Web server and closes it at the end of the session. We adopted the RUBiS bidding mix workload, and each client node has emulated 150 users.

3.3. Testbed

The environment used for the experimental tests is illustrated in Figure 3.

The System under Test consists of an application server (the Oracle Glassfish Server [2012]) running the benchmark applications. Only the application under test is active in the server. The user requests are simulated using several emulated clients. The *Sensor* module collects data and sends it to the *SHõWA* Application Performance Analyzer server for the analysis. This server contains the *Data Preparation*, *Workload Variation Analysis*, *Performance Anomaly Analysis*, *Anomaly Detector*, and *Root-Cause Failure Analysis* modules. To minimize the performance impact related to the transmission of the monitoring data, we considered using a dedicated network segment between the application server machine and the *SHõWA* Application Performance Analyzer server.

The test environment is composed of several machines. It includes four workload generator nodes with a 3GHz Intel Pentium CPU and 1GB of RAM each. The *SHõWA* Application Performance Analyzer machine includes two 2.66GHz Intel Xeon CPUs

and 2GB of RAM. The application server machine includes four 3.4GHz Intel Core i7 CPUs and 4GB of RAM, and the database server machine includes two 2.66GHz Intel Xeon CPUs and 2GB of RAM. All nodes run Linux with the 2.6 kernel version and are interconnected through a 100Mbps Ethernet LAN.

3.4. Anomaly Scenarios

To evaluate the ability of *SHõWA* to detect and pinpoint performance anomalies and distinguish performance anomalies from workload variations, we decided to test different types of anomaly scenarios:

—*Scenario A—CPU load*: In this scenario, the performance of the application is affected by a CPU contention. A CPU contention scenario can have multiple origins. Phenomenas like software aging [Garg et al. 1998] and unplanned or untested changes (OS or application upgrades, backup window, maintenance/system operations) can lead to CPU consumption scenarios potentially affecting the user-transaction response time. To impose load on the system, we used the `stress` tool [Stress 2010]. This is a very simple tool, and it was used to evaluate how *SHõWA* performs in the presence of a gradual CPU consumption with consequences for the user-transaction response time.
—*Scenario B—workload contention:* In this scenario, the number of user transactions begins to increase. A situation like this might occur due to a denial of service attack and may affect the performance of the application server queue, even leading to the rejection of new requests. Detecting scenarios of workload variation, with potential impact on the user-transaction response time or rejecting new requests, is of utmost importance to improving application performance and availability. For this test, we used *httperf* tool [Mosberger and Jin 1998]. This tool provides a flexible facility for generating various HTTP workloads. It was used to increase the number of user requests—between 50 and 1,500 new requests per second.
—*Scenario C—remote service change:* In this scenario, the database starts performing slowly and affects the application performance. The occurrence of ad hoc queries directly on the database without concerning their optimization or missing of indexes are examples of scenarios that may affect the performance of the database server and, consequently, the application behavior. We did a combination between miss of table indexes and inefficient queries to cause load on the database server.
—*Scenario D—memory consumption:* In this scenario, we simulate a memory leak. During the experiment, the amount of heap memory available in the application server is consumed at the rate of 1MB per second. When the amount of JVM memory is below a given threshold, the garbage collector becomes more aggressive in trying to reclaim memory, and may cause a performance degradation. If all memory is consumed, then an out-of-memory error is issued and the application server hangs. We used JAFL [Rodrigues et al. 2008], a Java fault load tool, to inject the anomaly.
—*Scenario E—application change:* In this scenario, we simulate a performance anomaly motivated by an application change. We have manually changed the application source code to include a function that sleeps for a certain amount of time. Rather than slowing down the response time continuously, this function intercalates periods of slowdown with periods of normal functioning. The response time increases by 3 milliseconds per minute.

According to the report presented in Pertet and Narasimhan [2005], these anomalies are common causes of software failures observed in Web applications.

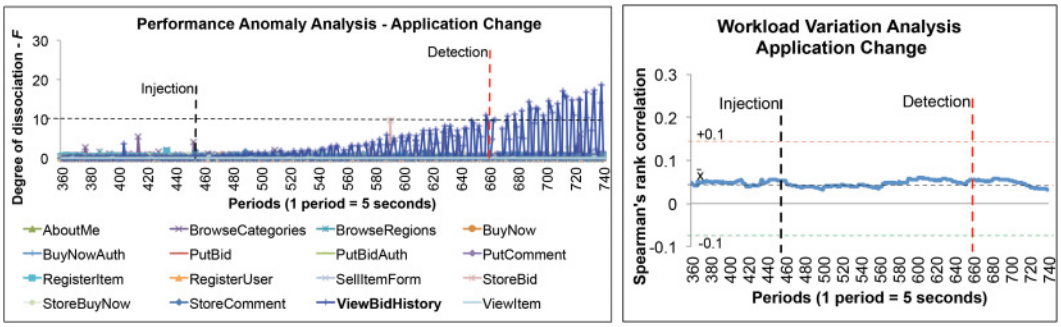


Fig. 4. Application change. (a) Degree of dissociation between the response time and the number of user transactions processed. (b) Correlation degree between the response time and the number of requests in the application server queue.

3.5. Detecting and Pinpointing Anomalies: Thresholds

The *Performance Anomaly Analysis* module detects a performance anomaly when the degree of dissociation is higher than 10. According to the analysis presented in Table II, a degree of dissociation higher than 10 corresponds to a significative response time delay that is not motivated by the load and mix of user transactions processed. From the table, it can be seen that a degree of dissociation higher than 10 includes situations where the response time delay is greater than 500 milliseconds, or even situations where the delay is just 100 milliseconds, but accompanied by a decrease in the workload. Detecting response time delays with this range of values is important, because according to Greg Linden,an increase of 500 milliseconds in the access latency to Google may result in a loss of 20% of its traffic [Linden 2006]. Similarly, an increase of just 100 milliseconds in the latency to Amazon may result in a reduction of 1% of sales.

A workload variation is detected when the *Workload Variation Analysis* module returns a variation of ρ (Spearman's rank correlation) higher than 0.1 degrees. Similarly, for the purpose of pinpointing anomalies (*Anomaly Detector* and *Root-Cause Failure Analysis* modules), we defined a Spearman's rank correlation variation higher than 0.1 degrees as the threshold value. The values were used in several experiments for evaluation of the *SHõWA* framework.

3.6. Detecting and Pinpointing Anomalies: Illustrating the Data Analysis Process

All anomalies were injected in both of the benchmark applications: TPC-W and RUBiS. From the results, we observed that the anomalies were detected and pinpointed by *SHõWA* in a similar manner. Since presenting all of the results largely increases the size of this article, we just illustrate the results for one of the anomalies injected: application change in the RUBiS benchmark application. This scenario covers all steps performed by the *SHõWA* framework to detect and pinpoint a performance anomaly.

In the presence of a performance anomaly, motivated by an application change, we expect that (1) the *Performance Anomaly Analysis* module is able to detect the user transaction affected by the performance anomaly; (2) the *Workload Variation Analysis* module does not reveal a problem with the workload execution; (3) the *Anomaly Detector* module does not identify any system or application server parameters related to the performance anomaly; and (4) the *Root-Cause Failure Analysis* pinpoints the application call, or set of calls, associated with the response time slowdown.

The results achieved by the *Performance Anomaly Analysis* module and the *Workload Variation Analysis* module are illustrated in Figure 4.

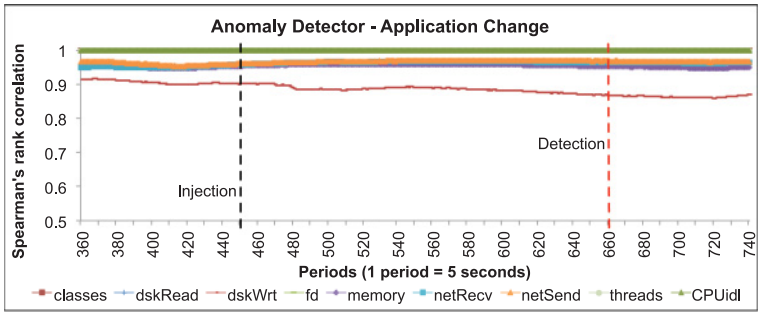
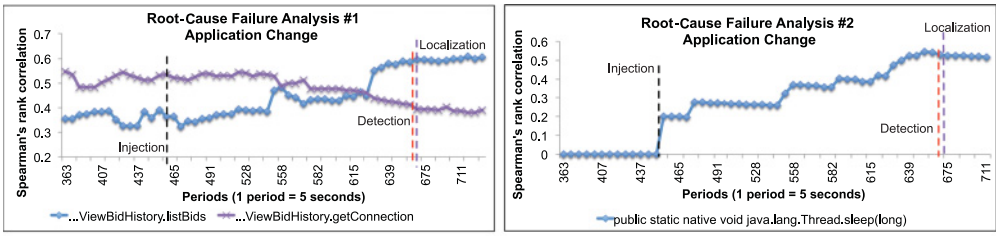


Fig. 5. Application change.



(a) (b)

Fig. 6. Application change. (a) Calls belonging to the `rubis_servlets_ViewBidHistory` user transaction associated with the response time slowdown. (b) Calls belonging to the `viewBidHistory.listBids` call associated with the response time slowdown.

The anomaly was injected around period 451. The results illustrated in Figure 4(a) reveal the existence of a response time slowdown affecting the *ViewBidHistory* user transaction. From the figure, the degree of dissociation between the response time and the number of user transactions processed becomes higher than 10 around period 661.

The analysis on a possible contention in workload is illustrated in Figure 4(b). The figure shows that between the anomaly injection and the time it was detected by the *Workload Variation Analysis*, the correlation degree between the response time and the number of requests in the queue almost did not vary. This indicates that the number of requests is not the main reason for the performance anomaly detected by the *Performance Anomaly Analysis* module.

The next step in the analysis consists of determining if there is a system or application server parameter associated with the performance anomaly identified in the previous analysis. This step is performed by the *Anomaly Detector*, and the results are illustrated in Figure 5.

The results presented in Figure 5 show no changes in the correlation degree between the parameters that are analyzed and the number of user transactions processed. In terms of overall analysis, it can be said that the performance anomaly is not related to changes at the system or application server level. The last step of the analysis is performed by the *Root-Cause Failure Analysis* module. This module evaluates the association between the response time of the calls belonging to the user-transaction call path and the response time of the user transaction. From the association degree, we intend to identify the calls related to the performance anomaly. The results relative to the `rubis_servlets_ViewBidHistory` user transaction are illustrated in Figure 6.

Figure 6(a) illustrates the set of calls that reported a high weight, in terms of response time, on the user transaction under analysis (`rubis_servlets_ViewBidHistory`). From

Table III. Percentage of End Users affected by a Slow Response Time Before, During, and After the Detection and Pinpointing of the Anomalies Injected (Average TPC-W and RUBiS)

	Response Time Delay (ms)	[S,I]	]I,D]	]D,P]	]P,E]
CPU Load	[100:500[	0.10%	0.37%	0.89%	12.55%
	[500:2000[	0.00%	0.00%	0.01%	7.91%
	[2000:∞[	0.00%	0.00%	0.00%	24.51%
Workload Overload	[100:500[	0.11%	0.82%	0.00%	0.63%
	[500:2000[	0.00%	0.00%	0.00%	28.94%
	[2000:∞[	0.00%	0.00%	0.00%	3.45%
Remote Change	[100:500[	0.18%	0.17%	0.24%	5.02%
	[500:2000[	0.00%	0.07%	0.81%	23.16%
	[2000:∞[	0.00%	0.78%	0.86%	11.51%
Memory Consumption	[100:500[	0.08%	0.02%	0.00%	1.78%
	[500:2000[	0.00%	0.01%	0.00%	0.24%
	[2000:∞[	0.00%	0.00%	0.00%	0.40%
Application Change	[100:500[	0.07%	0.14%	0.00%	7.83%
	[500:2000[	0.00%	0.02%	0.00%	0.00%
	[2000:∞[	0.00%	0.00%	0.00%	0.00%

the figure, it is verified that after the anomaly injection, the response time of the `viewBidHistory.listBids` call and the user-transaction response time becomes more correlated. Figure 6(b) illustrates the set of calls belonging to the `viewBidHistory.listBids` call path. From the list of calls, the analysis has identified a strong relationship between the `java.lang.Thread.sleep` call and the `rubis_servlets_ViewBidHistory` user transaction. The call matches the change made in the application source code.

3.7. Detecting and Pinpointing Anomalies: Results of the Experimental Study

The results presented in this section refer to the injection of the anomaly scenarios used in the experimental study.

The results are summarized in Table III by using the percentage of user requests affected by the response time slowdown. Results are presented in three distinct intervals. The first interval contains the percentage of transactions that, compared to the response time baseline previously measured, were affected by a response time delay between 100 and 500 milliseconds. The second interval contains the percentage of user transactions that suffered a response time delay between 500 and 2,000 milliseconds. The third interval presents the percentage of user transactions affected by a response time delay higher than 2,000 milliseconds. The results also consider four intervals of analysis: (1) $[S, I]$ contains the percentage of end users affected by the response time slowdown between Start of the experiment until the anomaly Injection period; (2) $]I, D]$ comprises the end users affected between the Injection and Detection periods; (3) $]D, P]$ corresponds to the percentage of end users affected during the Detection and Pinpointing of the performance anomaly period; (4) $]P, E]$ accounts for the users affected between Pinpointing until End of the experiment. Since we did not make any recovery attempt, a high percentage of end users is expected in the interval $]P, E]$.

From the percentage of end users affected by a response time slowdown shown in Table III, it can be seen that between the injection and detection (]I,D]) of the anomaly and between the detection and pinpointing (]D,P]), the percentage of user transactions affected is less than 1%. This means that *SHōWA* was able to detect and pinpoint the anomaly while the number of users affected was still low. In this experiment, we do not trigger any recovery action; therefore, the percentage of end users affected after the anomaly pinpointing period is high when compared to the previous periods. In one of the scenarios under study, in addition to the delay observed in response time, some

errors have occurred: after the memory consumption, there were 37 HTTP errors, and the application server had hanged. Such errors could be avoided if we had activated a recovery procedure.

4. *SHõWA*: ACCURACY ANALYSIS

In this section, we present an analysis of the accuracy achieved by the *SHõWA* framework for detecting and pinpointing anomalies. The analysis is presented at two levels.

At the first level, we check the following:

—*A: How many false positives occurred, given that there were no anomalies injected?*
—*B: How many anomalies were detected in the presence of anomalies (true positives)?*

At the second level of analysis, we check the following:

—*C: With respect to the result obtained in question A, what is the percentage of user transactions affected by a response time delay greater than 500 milliseconds?*
—*D: With respect to the result obtained in question B, what is the percentage of user transactions affected by a response time delay greater than 500 milliseconds?*

To answer questions A and C, we analyzed 24 hours of data. During these 24 hours, dynamic workloads totaling 3,784,982 requests were executed. We did not injected any type of anomalies. Considering the results, the answer to question A was 0%, and the answer to question C was 0.038%.

To answer questions B and D, we also analyzed 24 hours of data. During these 24 hours, dynamic workloads totaling 2,371,233 requests were executed. We injected five different types of anomalies: system overload, resource exhaustion (CPU and memory), and application changes. The anomalies were chosen taking into account their occurrence in Web applications (Causes of Failures in Web Applications [Pertet and Narasimhan 2005]). The anomalies were injected 10 times each, always under dynamic workloads. The moment of injection of the anomaly was randomly determined. According to the results, and specifically to that of question B, we observed that *SHõWA* was able to detect all anomalies injected. The answer to question D is separated in two periods: 15 minutes before the detection of the anomaly: 0.042%; 15 minutes after the detection of the anomaly: 0.71%.

These results highlight two important aspects: (1) *SHõWA* has detected all anomalies that we have injected, and when it detects the anomaly, it is because there are users being affected by the anomaly; (2) when there are no anomalies, *SHõWA* does not raise any false alarm.

5. *SHõWA*: PERFORMANCE IMPACT ANALYSIS

In this section, we present the results about the performance impact induced by *SHõWA*. To conduct this experiment, we started by determining the maximum capacity of the server in terms of response time and throughput. During this phase, there were no monitoring systems activated. After that, we performed tests to measure the performance impact induced by *SHõWA*, considering the use of the different adaptive algorithms: linear, exponential, and polynomial.

Each experiment was repeated 15 times. For each group of experiments conducted, we started with an execution using the TPC-W RBE followed by a test with the *httperf* tool. The *httperf* tool [Mosberger and Jin 1998] is a general-purpose tool that is useful to measure application and server limits. The application workload performed with the *httperf* tool mimics real end-user sessions, and it was executed by varying the number of concurrent user requests between 29 and 804 requests per second. According to the baseline results, the application server can process up to 517 requests per second until the response time is affected by a severe slowdown.

Table IV. Response Time Delay (ms) and Throughput Impact (%) Induced by *SHõWA* by Level of System Utilization and the Different Type of Adaptive Monitoring Algorithm Adopted

	Adaptation	0% to 25%	25% to 50%	50% to 75%	75% to 100%
Response Time Delay	Linear	0.00	0.00	0.42	3.40
	Exponential	0.00	0.00	0.40	2.80
	Polynomial	0.00	0.00	0.20	2.10
Throughput Impact	Linear	0.08%	0.09%	0.58%	4.14%
	Exponential	0.08%	0.10%	0.49%	3.62%
	Polynomial	0.04%	0.11%	0.24%	0.85%

To facilitate the analysis, the results were split into four intervals. These intervals correspond to the system utilization: 0% to 25% correspond to requests load varying between 29 and 144 requests per second, 25% to 50% correspond to requests load varying between 144 and 287 requests per second, 50% to 75% correspond to requests load varying between 287 and 431 requests per second, and 75% to 100% correspond to requests load varying between 431 and 517 requests per second.

The response time latency and server throughput is summarized in Table IV. The latency time is displayed as the difference in milliseconds considering the use of the *SHõWA* framework versus the baseline values. The throughput impact is presented as the percentage of user requests that the server was unable to process.

As presented in Table IV, the application-level monitoring (*SHõWA Sensor* module) does not induce a significant performance penalty. The response time delay induced while using the polynomial adaptation algorithm is only 2 milliseconds per request, and the throughput impact is less than 1%.

6. RELATED WORK

A self-healing system should be able to continuously gather information about itself and perform an analysis to detect the occurrence of anomalies. When an anomaly is detected and diagnosed, the system should be able to automatically select and execute a recovery procedure to avoid the harmful impact of failures. In the literature, we found numerous examples focusing and combining the tasks involved in a self-healing system for Web-based applications: monitoring, anomaly detection/localization, and recovery.

6.1. Monitoring Web-Based Applications

There are different monitoring techniques that are commonly adopted for Web-based applications. These techniques involve system-level, container-level, end-to-end, log analysis, and application-level monitoring systems.

System-level monitoring is used to keep track of several system parameters (e.g., CPU usage, memory utilization, number of open files). HP Operations Manager [2011], IBM Tivoli Monitoring [2012], Nagios [2009], and Zabbix [2010] are some examples of tools widely used by the industry. The application containers, like Web containers, usually include monitoring services that collect data of the application server parameters (e.g., JVM memory, thread pools, instanced objects, database connection pools, session details). This data is used by the IT staff to infer about the application server footprint and to become aware of adjustments to improve its performance. Tools such as Applications Manager [2014] and AppInternals Xpert [2014] are used to extract and consolidate the data from these monitoring facilities for detecting and diagnosing problems. End-to-end is a type of monitoring used to capture the user perspective about the service—that is, to know if users are having issues while accessing the Web application. It is an application monitoring solution that typically behaves like a synthetic user and allows collection of data to help IT operators identify and resolve issues before they

impact the real users. Among many end-to-end monitoring examples, we found leading solutions such as Gomez [2014] and Site 24x7 [2010].

Log files contain information such as system operations, application server operations, network performance, input/output performance, and application and system warnings and error messages. These records are quite important for anomaly inference and are often used for troubleshooting. An interesting work based on log file monitoring is presented in Bodíc et al. [2005]. The authors have used the historical HTTP log files to model the access patterns to each Web page and alert the sysadmins when the patterns change.

Application-level monitoring is gaining expression within Web applications. This type of monitoring complements the previous ones. For example, in addition to indicating if a transaction is slow, it can be used to indicate whether the slowness is due to the state of the resources or due to application changes. A sound work that makes use of application-level monitoring is the Pinpoint project [Chen et al. 2002]. This project relies on a modified version of the application server to record the request execution paths and build a model of the expected interactions between components, without requiring prior knowledge about the application. At runtime, it measures the differences between the expected and observed interactions. The authors have noticed that Pinpoint introduces an overhead of about 8.4%. Another interesting work is proposed by Cherkasova et al. [2008].The authors adopt bytecode instrumentation to collect the transactions latency and count the number of transactions, the number of outbound calls, and the average latency of the outbound calls. This data is used to build a regression-based transaction model and an application signature that is then used to detect anomalies and analyze performance changes in the application.

The *SHõWA* framework performs application-level monitoring. It performs bytecode instrumentation and does not require manual changes to the application source code to collect data. From the bytecode-based systems presented earlier, the biggest advantage of *SHõWA* has to do with the data it collects without inducing a significant performance overhead. *SHõWA* measures the execution time of the application transactions and collects data from the system, from the application server, and from the application at runtime. It includes adaptive algorithms that allow profiling of the application calls with low performance impact: the response time delay is just 2 milliseconds per request, and the throughput impact is less than 1%. Due to the large number of parameters involved in a Web environment, performing fine-grain monitoring is extremely important to pinpoint the failures in a timely manner.

6.2. Anomaly Detection and Localization in Web-Based Applications

In the literature, we have found several research projects focusing on the automatic detection and pinpointing of anomalies in Web-based applications. The approaches followed by those projects can roughly be divided into three types: (1) those attentive to the usage pattern, (2) those responsive to error manifestation, and (3) those watchful of performance variations.

The usage patterns analysis aims to observe how the system or service state differs from the expected usage. The type of analysis is commonly based on time series models, machine learning, and structural models—that is, models watchful to changes such as a different execution path taken by the transactions. Some of the most relevant projects that we found in the context of Web applications adopting this approach are Li et al. [2002], Bodíc et al. [2005], Candea et al. [2006], and Kiciman and Fox [2005].

Error manifestation and error propagation analysis is another approach that has been explored by several authors (e.g., Li et al. [2007], Candea et al. [2003], Bellur and Agrawal [2007], and Carzaniga et al. [2010]) to detect and pinpoint the components responsible for failures in Web-based applications.

Detecting structural changes and modeling the error manifestation are interesting approaches, but they are unable to detect scenarios of low performance. In this context, tracking a *customer-affecting metric* (e.g., response time) is useful for monitoring the service. Along with the customer-affecting metric, tracking system or environmental metrics enhances the detection and anomaly localization ability. The work presented in Cherkasova et al. [2008] is an example where the performance of the service is taken into consideration to detect performance anomalies. The authors combine a regression model and an application signature to identify if a response time variation is due to a workload change, an application change, or an anomaly. The regression-based model is used to model the CPU demand of the application transactions across different time segments. The anomalous segments describe the scenario where the observed CPU utilization cannot be explained by the application workload. During the modeling phase, these segments are deliberately removed from the regression-based model to avoid corruption on the regression estimations of the segments with normal behavior. The application performance signature is used to uniquely reflect the application transactions and their CPU requirements independently from the workload types. Whereas the regression-based approach is useful to accurately detect a difference in the CPU consumption model of application transactions and alarms about a performance anomaly or a possible application change, the application performance signature allows comparison of the new application signature to the old one and detects which of the transactions was affected by a performance change. The Magpie framework [Barham et al. 2003] is also an interesting work in the area. It adopts fine-grain analysis to characterize the transaction resource footprints in fine detail. Then, clustering and probabilistic state machine techniques are used to observe performance changes and identify which events or event sequences are behind an anomalous behavior. Preliminary results presented by the authors shown that Magpie has good potential; however, extra work needs to be done, particularly to deal with high intrarequests concurrency.

The data analysis included in the *SHõWA* framework fits the performance variation analysis approach. From the preceding works, one more related to our approach is found in Cherkasova et al. [2008]. *SHõWA*is different in that it allows one to detect if a given performance anomaly is due to a workload change, a system change, an application server change, or an application change. It also pinpoints the parameters behind these changes, contributing to the selection of appropriate recovery procedures. Unlike Magpie, *SHõWA* does not induce a significant performance impact on the system. *SHõWA* makes use of adaptive and selective monitoring, thus reducing its own performance overhead and without losing the ability to profile the application under monitoring.

6.3. Failure Recovery

A survey presented in Schneider et al. [2014] divides the recovery methodologies in three types: supervised, semisupervised, and unsupervised. The recovery is more autonomous as it moves from a fully supervised to an unsupervised mode.

The recovery methodologies following the supervised approach are the most common. They allow the execution of a validated and controlled set of recovery actions, reducing the uncertainty in system behaviors. The projects presented in Garlan et al. [2004] and Pernici [2008] are two examples where the supervised methodology is adopted for recovery.

The semisupervised methodology makes use of SLAs or utility functions to select a recovery procedure. For example, the Shadows framework [Shehory 2006] combines historical data with utility functions to define the recovery actions. These actions are then revalidated by a human operator.

In the unsupervised methodology, the recovery actions are defined, selected, and executed without human interaction. In Ramirez et al. [2011] and Carzaniga et al. [2008], the authors present unsupervised approaches that upon the occurrence of problems automatically drive the evolutionary process toward new viable solutions.

As in Garlan et al. [2004] and Pernici [2008], the recovery of anomalies provided by *SHõWA* is done by a supervised methodology. *SHõWA* includes recovery procedures and high-level policies that control the recovery process. The data on the implementation of the recovery procedures is being stored so that in the near future, it will be possible to change the methodology to a semisupervised methodology.

7. CONCLUSION

In this article, we presented a framework targeted for the detection, localization, and recovery of performance anomalies in Web-based applications. The framework allows collection of application-level data without changing the application source code. It performs statistical analysis to detect and pinpoint performance anomalies independently from the workload variations. The framework is generic and can be applied to any type of transactional Web application. The only module that needs to be ported is the *Sensor* module. Fortunately, AOP libraries for most Web containers exist (e.g., PHP, .Net, J2EE), making this task easy to accomplish.

Besides the presentation of the framework, we conducted an experimental study to evaluate the ability of *SHõWA* to detect and pinpoint different types of anomalies across two benchmark applications: one retail store Web application and an auction Web application. The applications were exercised through highly dynamic workloads. Taking into account the implementation of the *SHõWA* framework and the experimental results achieved so far, we highlight the following:

—By adopting Spearman's rank correlation, it is possible to measure the correlation between the variables independently from its scale. This allows generalization of the analysis to different applications and the definition of a single threshold value for all user transactions.
—*SHõWA* was able to detect and spot all anomalies that we injected, and it has distinguished them from workload variations.
—Anomalies were detected and pinpointed while the number of users affected was low.
—When there is a very dynamic workload but there are no faults, the tool was able to detect this situation and did not raise any false alarms, which was an important result.
—With the adoption of adaptive and selective monitoring algorithms, the server throughput was affected in less than 1%, and the response time penalty per request varied between 0.5 (server load below 75% of its maximum capacity) and 2 milliseconds (server load above 75% of its maximum capacity).

The ability to detect anomalies while the number of end users affected is low and the accuracy results make the *SHõWA* framework a key tool for the automatic detection and recovery of performance anomalies in Web-based applications. Considering the importance of Web applications and the negative consequences of their failures to businesses, the adoption of a tool such as *SHõWA* is extremely important to reduce malfunctions in these systems, maximizing their availability and performance.

REFERENCES

Aberdeen Group. 2010. Web Performance Today. http://www.webperformancetoday.com/2010/06/15/everything-you-wanted-to-know-about-web-performance.

AppInternals/SteelCentral. 2014. Riverbed Application Performance Management. http://www.riverbed.com/products/performance-management-control/application-performance-management.

Applications Manager. 2014. Application performance monitoring tool. http://www.manageengine.com/products/applications_manager/.

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2001. Fail-stutter fault tolerance. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*. 33–38.

Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems*. 15.

Umesh Bellur and Amar Agrawal. 2007. Root cause isolation for self healing in J2EE environments. In *Proceedings of the 1st International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, Los Alamitos, CA, 324–327. DOI:http://dx.doi.org/10.1109/SASO.2007.46

Peter Bodíc, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jonathan Hui, Armando Fox, Michael I. Jordan, and David A. Patterson. 2005. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proceedings of the International Conference on Autonomic Computing*. IEEE, Los Alamitos, CA, 89–100.

George Candea, Mauricio Delgado, Michael Chen, and Armando Fox. 2003. Automatic failure-path inference: A generic introspection technique for Internet applications. In *Proceedings of the 3rd IEEE Workshop on Internet Applications*. 132.

George Candea, Emre Kiciman, Shinichi Kawamoto, and Armando Fox. 2006. Autonomous recovery in componentized Internet applications. *Cluster Computing* 9, 2, 175–190.

Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. 2010. Automatic workarounds for Web applications. In *Proceedings of the ACM SIGSOFT 18th Symposium on Foundations of Software Engineering (SIGSOFT 2010/FSE-18)*. 237–246.

Antonio Carzaniga, Alessandra Gorla, and Mauro Pezze. 2008. Healing Web applications through automatic workarounds. *International Journal on Software Tools for Technology Transfer* 10, 6, 493–502.

Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. 2002. Performance and scalability of EJB applications. *ACM SIGPLAN Notices* 37, 11, 246–261.

Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*. 595–604.

Ludmila Cherkasova, Kivanc M. Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. 2008. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Proceedings of the International Conference on Dependable Systems and Networks*. 452–461.

Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences* (2nd ed.). Lawrence Erlbaum.

Alan G. Ganek and Thomas A. Corbi. 2003. The dawning of the autonomic computing era. *IBM Systems Journal* 42, 1, 5–18.

Sachin Garg, Aad Van Moorsel, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. 1998. A methodology for detection and estimation of software aging. In *Proceedings of the the 9th International Symposium on Software Reliability Engineering*. 283–292.

David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 37, 10, 46–54.

Gomez. 2014. Compuware Application Performance Management solution. http://www.ndm.net/apm/Compuware/gomez.

HP Operations Manager. 2011. Fault and Performance Monitoring. Available at http://www8.hp.com/us/en/software-solutions/software.html?compURI=1170678.

IBM Tivoli Monitoring. Proactive Monitoring. Available at http://www-03.ibm.com/software/products/us/en/tivomoni/.

Terence Kelly and Alex Zhang. 2006. *Predicting Performance in Distributed Enterprise Applications*. Technical Report HPL-2006-76. HP Laboratories, Palo Alto, CA.

Emre Kiciman and Armando Fox. 2005. Detecting application-level failures in component-based Internet services. *IEEE Transactions on Neural Networks* 16, 5, 1027–1041.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*. 220–242.

Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4, 3, 382–401.

Junguo Li, Gang Huang, Jian Zou, and Hong Mei. 2007. Failure analysis of open source J2EE application servers. In *Proceedings of the 7th International Conference on Quality Software*. 198–208.

Lei Li, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. 2002. An approach for estimation of software aging in a Web server. In *Proceedings of the International Symposium on Empirical Software Engineering*. 91–100.

Greg Linden. 2006. Make Data Useful by Greg Linden, Amazon.com. Retrieved January 29, 2015, from http://www.scribd.com/doc/4970486/.

David Mosberger and Tai Jin. 1998. Httperf—a tool for measuring Web server performance. In *Proceedings of the 1st Workshop on Internet Server Performance*. 59–67.

Nagios. 2009. IT Infrastructure Monitoring. Available at http://www.nagios.org/.

Oracle Glassfish Server. 2012. Java Application Servers. Available at http://www.oracle.com/technetwork/middleware/glassfish/.

Barbara Pernici. 2008. Self-healing systems and Web services: The WS-Diamond approach. In *Business Process Management Workshops*. Lecture Notes in Business Information Processing, Vol. 17. Springer, 440–442.

Soila Pertet and Priya Narasimhan. 2005. *Causes of Failure in Web Applications*. Technical Report. Parallel Data Lab, Carnegie Mellon University, Pittsburgh, PA.

Sean Power. 2010. Metrics 101: What to Watch. Retrieved January 29, 2015, from http://www.slideshare.net/bitcurrent/metrics-101.

Andres J. Ramirez, David B. Knoester, Betty H. C. Cheng, and Philip K. McKinley. 2011. Plato: A genetic algorithm approach to run-time reconfiguration in autonomic computing systems. *Cluster Computing* 14, 3, 229–244.

Carroll Rheem. 2010. Consumer response to travel site performance. In *A PhoCusWright and Akamai WHITEPAPER*.

Nuno Rodrigues, Décio Sousa, and Luis Silva. 2008. A fault-injector tool to evaluate failure detectors in grid-services. In *Making Grids Work*. Number 978-0-387-78447-2. Springer, 261–271.

Chris Schneider, Adam Barker, and Simon Dobson. 2014. A survey of self-healing systems frameworks. *Software: Practice and Experience*. DOI:http://dx.doi.org/10.1002/spe.2250

Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22, 4, 299–319.

Onn Shehory. 2006. A self-healing approach to designing and deploying complex, distributed and concurrent software systems. In *Programming Multi-Agent Systems*. Lecture Notes in Computer Science, Vol. 4411. Springer, 3–13.

Bojan Simic. 2010. Ten Areas That Are Changing Market Dynamic in Web Performance Management. Available at http://www.trac-research.com/web-performance/.

Site 24x7. 2010. Website Monitoring. http://www.site24x7.com.

Wayne D. Smith. 2001. TPC-W: Benchmarking an Ecommerce Solution. Available at http://www.tpc.org/tpcw/.

Stress. Load and Stress Test Tool. http://linux.die.net/man/1/stress.

Zabbix. 2010. Enterprise Monitoring Solution. http://www.zabbix.com/.

Jerrold H. Zar. 1972. Significance testing of the Spearman rank correlation coefficient. *Journal of the American Statistical Association* 67, 339, 578–580.

Received February 2014; revised September 2014; accepted October 2014