Technique for Detecting Early-Warning Signals of Performance Deterioration in Large Scale Software Systems

Raghu Ramakrishnan[*]
Tata Consultancy Services
Noida, UP, INDIA
raghuramakrishnan71@gmail.com

Arvinder Kaur[†]
USICT, Guru Gobind Singh Indraprastha University
Dwarka, Delhi, INDIA
arvinder70@gmail.com

ABSTRACT

The detection of early-warning signals of performance deterioration can help technical support teams in taking swift remedial actions, thus ensuring rigor in production support operations of large scale software systems. Performance anomalies or deterioration, if left unattended, often result in system slowness and unavailability. In this paper, we presents a simple, intuitive and low-overhead technique for recognizing the early warning signs in near real time before they impact the system The technique is based on the inverse relationship which exists between throughput and average response time in a closed system. Because of this relationship, a significant increase in the average system response time causes an abrupt fall in system throughput. To identify such occurrences automatically, Individuals and Moving Range (XmR) control charts are used. We also provide a case study from a real-world production system, in which the technique has been successfully used. The use of this technique has reduced the occurrence of performance related incidents significantly in our daily operations. The technique is tool agnostic and can also be easily implemented in popular system monitoring tools by building custom extensions.

Keywords

System monitoring, performance anomaly, performance deterioration, Little's law, production systems, closed systems

[*]Raghu Ramakrishnan is Technology Head of Government Business in Tata Consultancy Services, Noida, INDIA. He is also pursuing research at USICT, Guru Gobind Singh Indraprastha University, Delhi, INDIA.

[†]Arvinder Kaur is Professor in University School of Information and Communication Technology, Guru Gobind Singh Indraprastha University, Delhi, India.

ICPE'17, April 22-26, 2017, L'Aquila, Italy
© 2017 ACM. ISBN 978-1-4503-4404-3/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3030207.3044533

1. INTRODUCTION

Large scale, heterogeneous software systems are becoming pervasive in many areas. Such systems are usually not monolithic and may be made of a number of constituent systems working in an integrated manner to fulfil customer requirements. Monitoring the performance of critical software systems at run-time is an important activity carried out during the operations phase of the software life cycle. The main objective of performance monitoring is to be able to proactively identify performance related deterioration or anomalies much before they severely affect the end users. We can describe a performance anomaly in a software system as an unanticipated, undesirable degradation or deterioration in the performance of the system at run-time. Performance anomalies can severely impact services provided by the system to the customers, resulting in customer complaints, loss of business in the form of potential customers and penalties on account of violating contractual service level agreements. The availability of a functional system monitoring setup is an essential prerequisite for operational readiness and go-live of large scale software systems. System monitoring can be carried out at different levels of granularity like continuous monitoring, monitoring at predefined time (e.g. analyzing and reporting the system health at close of business) or post-hoc monitoring (e.g. reporting investigation results after occurrence of an event or incident in production). Early, continuous and near-real time detection of performance deterioration signals help Network Operations Center (NOC) teams to proactively carry out remediation actions, increasing the chances of avoiding system slowness or unavailability. NOC comprises of one or more command and control centers from where the production system is monitored to ensure its smooth functioning.

Open source and commercial available system monitoring tools, as they are commonly called in the IT industry have matured over time and established their place in IT monitoring landscape. The basic version of system monitoring tools collect a large number of performance counters or metrics from the components on which software systems are deployed (like physical servers, virtual machines, networks, storage, web server, application server, database servers) at predefined time intervals. Some of the monitoring tools can also be configured to collect and display metrics related to web services, like response time and throughput. The detection and alerting of performance deterioration in these system monitoring tools is mostly based on identifying performance counters which exceed static or dynamic threshold values set by administrators.

Often, the components of critical large scale software systems are configured for high availability using a N+M redundancy model. In an N+M model, N components required for full functioning have a minimum of M redundant components. For critical systems, it is important to note that the immediate priority on detecting a performance deterioration is to take actions for preventing or reducing any further impact on the Quality of Service (QoS) offered by the system. Root Cause Analysis (RCA) is is a secondary objective and is expected to follow later. The time available between the detection of a performance anomaly and it transformation to a major performance incident is very limited. Therefore, any technique designed to detect anomalous conditions that emerge during operations should not rely on triage and analysis of a large number of IT infrastructure performance counters. Our experience in the industry tells us that the common remediation actions carried out by support teams include killing and restarting application processes, servers, terminating expensive database queries and locks. If the components are redundant, they can be restarted one at a time to minimize downtime. The optimization or fixing of code, tuning of parameters, realigning resources, augmentation of compute, storage or network needed to overcome performance problems, is usually done on a short to medium term horizon.

Our extensive experience in the area of monitoring large scale software systems and running NOC operations indicates that ease of implementation, intuitiveness and low runtime overhead of an anomaly detection technique is a driving factor in its effectiveness and adoption. The paper describes a technique based on application services instead of a large number of IT infrastructure performance counters for identifying early-warning signals of performance deterioration, which we have been successfully using in production support of mission critical software systems for last few years. The technique has helped us to proactively undertake remediation activities and eliminate occurrences of high severity performance incidents. The technique, which initially relied on continuous visual inspection to match steep increase in average system response time with an abrupt decline in system throughput, was enhanced to have automated detection capabilities through the use of Individuals and Moving Range (XmR) control charts.

The contributions of this paper are:

- We propose a technique based on Little's Law and XmR control charts to identify early-warning signs of performance deterioration. The technique is simple, intuitive and has a low overhead.

- Software systems are usually deployed using a multi-tier architecture like presentation, application and data. The proposed technique is deployment tier agnostic i.e. it is equally applicable for all the tiers.

- We demonstrate that our technique can automatically detect early-warning signs of performance deterioration by evaluating its accuracy on a large software system used in industry.

The paper is organized as follows: Section 2 provides a motivating example driving this work. Section 3 explores the related work in this area. Section 4 describes our technique. Section 5 presents a reference implementation of the technique. Section 5 details real-world case study of performance anomaly detection during production support. Section 6 lists the threats to the validity of our work. Section 7 provides the conclusions and direction on future work.

2. MOTIVATING EXAMPLE

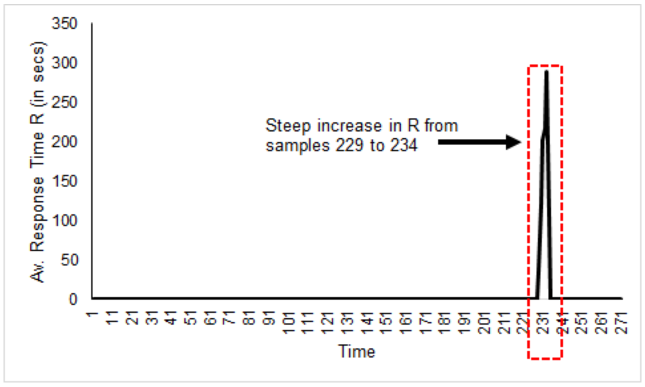
In this section, we describe an actual situation observed by the production support team while monitoring the performance of a large scale case management system. The system is used for processing application forms daily at various customer touch points. The system has a three tier deployment architecture: web, application and data tier. The web tier contains the presentation components, application tier contains web services and data tier hosts the database, document store. The throughput and average response time at each tier is calculated from the data collected from the web tier log, application tier log and database snapshot. The throughput is represented by the symbol X and the average response time is represented by the symbol R. For web and application tier, the standard web server logs are used. Throughput is the number of requests processed by the software system in a given time interval. Response time is the time elapsed between the software system receiving the request and the requests is processed. The terms response time and processing time are used interchangeably in this paper. The value of X and R are calculated every two minutes. Figures 1, 2 and 3 shows the value of X and R plotted against time. The total number of measurements is 270, conforming to a monitoring duration of nine hours. We now examine the graphs related to each tier in detail.

Web Tier: In Figure 1a, an abrupt increase in R is observed in monitoring intervals 229 to 234. The average processing time R changed from 0.47 s to 34.49 s. The increase is accompanied by a rapid decline in X in the same interval, as observed in Figure 1b. The throughput dropped from 2,215 to 58. Figures 1c and 1d show a magnified view of the monitoring intervals 226 to 237 for better understanding.

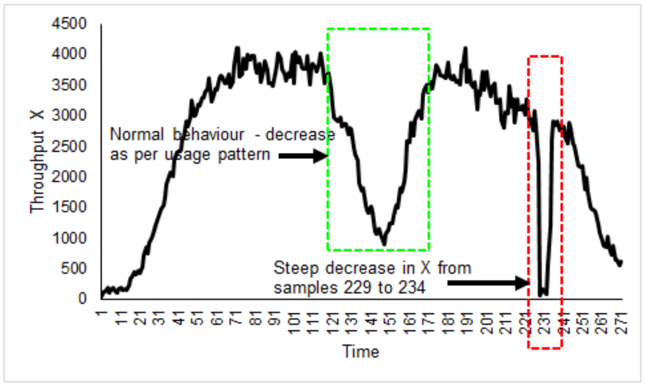
Application Tier: In Figure 2a, a steep increase in R is observed in monitoring intervals 229 to 234. The average processing time R changed from 0.36 s to 75.69 s. The increase is accompanied by a rapid decline in the throughput X in the same interval, as observed in Figure 2b. The throughput dropped from 2,431 to 173. The web and application tier show a similar pattern for throughput because the web tier uses the facade design pattern and makes a single call to composite web services in the application tier. However, the web services may internally invoke a number of database queries for completing a functionality. The same can be observed in the large value of X in Figure 3b. The maximum value of X is more than 188,000.

Data Tier: The average execution time for database queries collected from the database snapshot during normal operations, was between 10 ms to 20 ms. Figure 3a shows an increase in R to 0.099 s. The increase is accompanied by a rapid decline in the throughput X in the same interval, as observed in Figure 3b. The throughput dropped from 123,000 to 91,000.

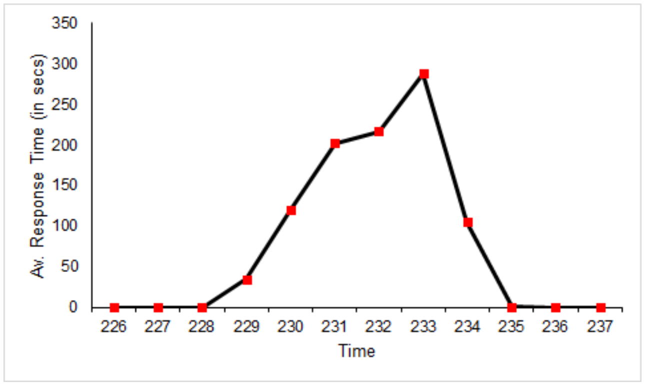
The performance incident in this case, continued till the end of monitoring period 234. The root cause in this incident was a database query with an incorrect filter criteria returning a large number of records to the calling web service, leaving



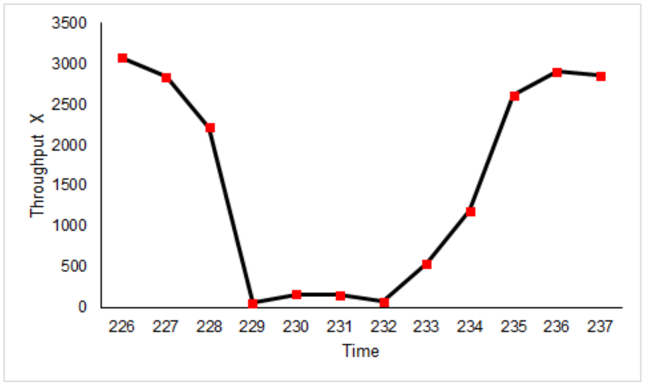
(a) R vs. time



(b) X vs. time

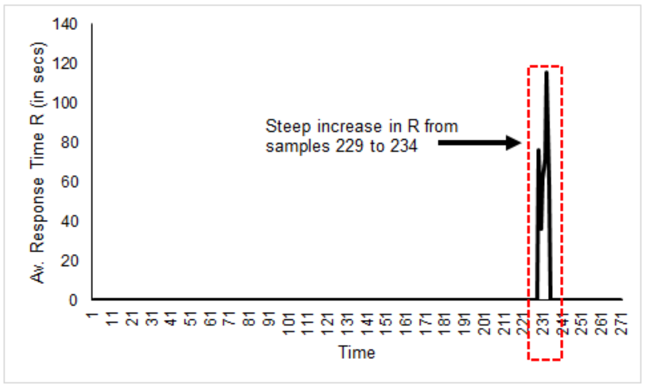


(c) R during performance deterioration

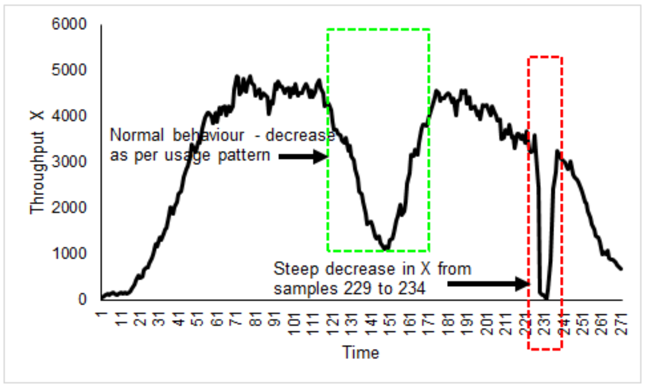


(d) X during performance deterioration

Figure 1: X and R for web tier



(a) R vs. time



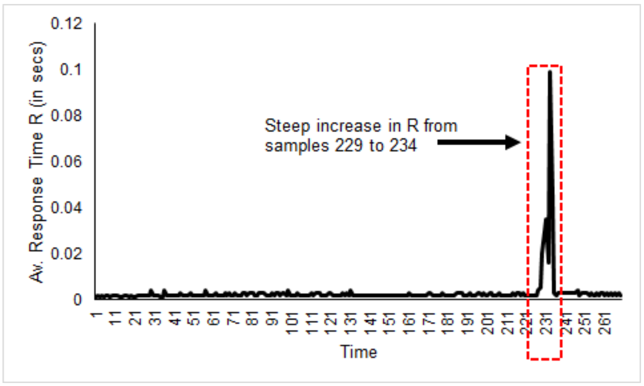
(b) X vs. time

Figure 2: X and R for application tier

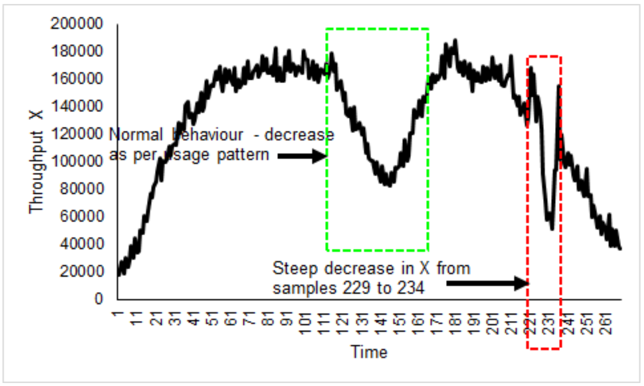
the application server with no free memory. The threads which were currently executing web services in the application server stalled causing the calling web tier services to become extremely slow. The early-warning signals of an impending performance related incident are clearly seen in sampling period 229 and 230. The signals can be easily detected by continuous visual monitoring of the pair of R and X values, for each tier. While, using visual analysis, we can detect such patterns which point to a performance anomaly or abnormal performance behaviour, it is prone to misses. Our objective is to design a technique to detect such patterns automatically. Such a technique will enable generation of early warning alerts so that remediation can be quickly carried out.

3. RELATED WORK

In this section, we review the available work related to identifying performance related anomalies in software systems. The anomaly detection techniques can be classified as online or offline. Online techniques analyze monitoring data as it gets collected. Offline techniques analyze moni-



(a) R vs. time



(b) X vs. time

Figure 3: X and R for data tier

toring data at specified time e.g. at the end of load testing execution cycle, after business hours.

There are a number of techniques in literature for detecting anomalies during performance monitoring of production systems or at the completion of a load test. Malik et. al. recommend use of supervised and unsupervised learning techniques like K means clustering, principal component analysis, WRAPPER to spot anomalies in IT infrastructure metrics collected during load test execution [19][18][16][17]. For checking performance related regressions in a new release of a software, Nguyen et. al. use statistical control charts [21]. The control charts are based on IT infrastructure metrics like CPU utilization, memory utilization, network IO and disk IO. The distribution of response time from a given load test can be compared with the distribution of response time from a baseline test. If the distributions are not similar, then the load test results need to be investigated further for performance anomalies [12]. The load testing results are validated at the end of the test using Little's Law [20].

Frank et. al. provide a detailed comparison of Hypothesis Testing, Statistical Process Control, Multivariate Adaptive Statistical Filtering (MASF) and ANOVA techniques used for detecting abnormality in metrics that are caused by assignable reasons [7]. The use of control charts, MASF and its variations for monitoring software systems was proposed by Trubin et al. [24][25][26][27]. MASF partitions the time during which the system is operational, into hourly, daily or weekly reference segments to characterize repeatable or similar workload behavior experienced by a software system [8]. For example, the workload encountered by the system on Monday between 9:00 a.m. - 10:00 a.m. may be different from the workload between 10:00 a.m. - 11:00 a.m. Each segment is characterized by its mean and standard deviation. The number of reference sets can be further reduced using clustering techniques. The upper and lower limits are established for each reference at three standard deviations from the mean.

Foo et. al. propose the use of association rule mining technique on metrics from a baseline load test run to establish rules, with each rule containing a set of metrics which are correlated [11]. For example, a large number of requests may correlate with high CPU utilization. The violations of defined rules is taken as an early-warning sign of a performance anomaly. The use of predictive rules generated by an offline analysis of historical incident tickets and alerts in system monitoring is described in [23]. Satoshi et. al. use control charts with average, maximum, median and minimum statistic to proactively identify response time anomalies in web applications [13]. João Paulo et. al. define a framework comprising of aspect-oriented programming (AOP) based application performance data collection, statistical correlation and time-series alignment techniques to identify the incidence of performance anomalies [15]. Performance signatures established using failures which have occurred in the past can also be used to proactively determine anomalies [1, 5, 11]. Cherkasova et. al. propose an integrated framework comprising of a regression based transaction model and an application performance signature for enabling service providers to quickly identify and prevent performance related problems [9].

Open source and commercial system monitoring tools collect various performance data at defined intervals. The collected data is then compared to one or more threshold values for detecting performance issues. Thresholds can be simple (compare a single observation to the threshold), average (compare the average of most recent n observations to the threshold), successive (threshold is violated multiple consecutive times), occurrences (threshold is violated m times in n observations) or delta (of two observations separated by a specified number of observations violates the threshold) type [4]. The tools allow configuration of separate warning and critical threshold values. It is also possible to set the number of consecutive times a threshold must be violated for triggering an alert.For example, if CPU is 90% or more for over 3 times, an alert can be triggered. The threshold values can be defined for different times of the day or week to account for variations in workload profile. Apart from manually setting the thresholds, the tools also provide adaptive [6] or self-tuning [4] methods. In this case, the thresholds are statistically calculated periodically using historical data. The threshold values are based on statistic like percentile, a percentage of maximum value or a scaling factor of standard deviation.
System monitoring tools allow for searching for patterns and keywords in logs that may characterize a performance (or functional) issue. Some of the tools allow for deriving the health of a monitored entity based on the health of its constituents [3]. For example, we can define the overall hard-

ware availability of a system to be dependent on the availability of all its server clusters. The availability of a server cluster may further depend on the availability of individual servers in the cluster.

Of all the above techniques, association rules and statistical process control charts appear to be the most intuitive for practitioners and easy to implement in commercial or open source system monitoring tools using custom extensions. However, association rules require thresholds to be set, which is dependent on the knowledge and experience of the practitioner. Similarly, a control chart can be used to monitor a single variable at a time, which restricts our ability to correlate multiple variables of interest. Through this work, an attempt has been made to correlate the monitoring done by two separate control charts: one for system throughput and the second for average system response time. The correlation is based on the inverse relationship between system throughput and average system response time as established by Little's Law [12],

4. PROPOSED TECHNIQUE

Software systems can be modeled using an open, closed or partly-open system model [22]. In a closed system model, a user submits a new request only when the response for the previously submitted request is received. This model is characterized by a fixed number of users. Load testing of software systems involving a fixed number of virtual users executing scripts containing data parametrization is an example of closed system behaviour. In an open system model, request submissions and completions are independent. Each arriving user submits a request, receives the response and then departs from the system. This model is characterized by the average arrival rate of users (λ).

In real-world, software systems are usually a mix of both open and closed models. Such systems are called as hybrid or partly-open systems as shown in Figure 4. Box 1 is the boundary of a partly-open system comprising of user arrivals, request-response interactions and departures. Box 2 is the boundary of a closed system containing only request-response interactions of users. The users arrive at and depart from the software system like in an open system. Prior to departing, users interact with the system multiple times. During every interaction, the user submits a request, waits to receive the response before submitting the next request. Partly-open systems can be treated as a closed system if the number of requests in user sessions are $\geq$ 10 [22]. So, for all practical purposes, we can model real-world software systems as closed systems.

For closed systems, Little's Law [14] establishes the relationship between the number of users in the system (N), throughput of the system (X), average response time (R) and average think time of the user (Z). The relationship is described by Equation (1). The terms X and R have been already introduced in section 2. Think time is the time spent by the user between consecutive interactions with the system like entering data in a form, viewing results of a previous request, navigating on the screen [28].

$$N = X(R + Z) \qquad (1)$$

For closed systems, it can be intuitively argued that because N and Z do not change, X is inversely related to R [12]. For open systems, because N is not fixed, X and R are

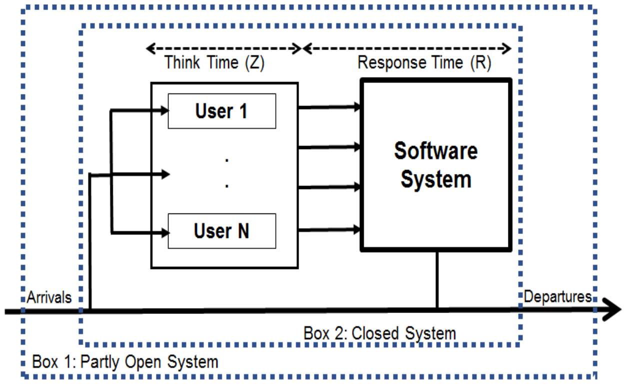


Figure 4: Partly open model representation of a software system

not correlated. For illustration, let us consider an interactive software system with 20 users, average think time of 5 sec and average response time of 15 sec. The throughput of the system calculated using Equation (1) is $\frac{20}{(15+5)} = 1$ request/sec. If the average response time increases by two times to 30 sec, the throughput of the system decreases to $\frac{20}{(30+5)} = 0.57$ request/sec. The increase in average response time by 100% leads to 43% decrease in throughput. Our experience shows that commonly used system monitoring tools in industry have no in-built support for establishing such correlation.

The proposed technique leverages the inverse relationship between X and R for identifying early-warning signals of performance deterioration in software systems. For this, we consider monitoring of software systems in production as a simplified process with two variables: throughput (X_S) and average response time (R_S). The subscript S denotes the monitoring interval. X_S is the number of requests completed in interval S. R_S is the sum of processing times of all requests completed in interval S divided by X_S. The value of X_S and R_S is calculated for requests processed during interval S. For example, a software system processed 400 requests in an interval of two minutes. The sum of the individual processing times of all the requests is calculated as 32 sec. Using this data, X_S and R_S are calculated as 400 and $\frac{32}{400} =$ 0.08 sec respectively. The value of X_S and R_S can increase or decrease from their values of the preceding monitoring interval (S-1). Figure 5 shows the four scenarios which can arise while comparing consecutive X_S and R_S values. Each scenario corresponds to one of the four quadrants. The X axis represents the behavior of X_S and the Y axis represents the behaviour of R_S.

As all pairs of X_S and R_S which fall in quadrant (1) may not be a real-warning signal, our objective is to devise a mechanism to distinguish signals from noise. For this, we will use the Individuals chart of an Individuals and Moving Range (XmR) statistical process control chart. The XmR chart is a set of two charts: X and mR. The mR chart shows the moving ranges of the variable of interest calculated using Equation (3) and the X or Individuals chart shows the individual values [10]. We use a two-point moving average. The Central Line (CL), Upper Control Limit (UCL) and Lower Control Limit (LCL) of the Individuals chart is calculated using Equations (4) to (6). The CL and UCL for correspond-

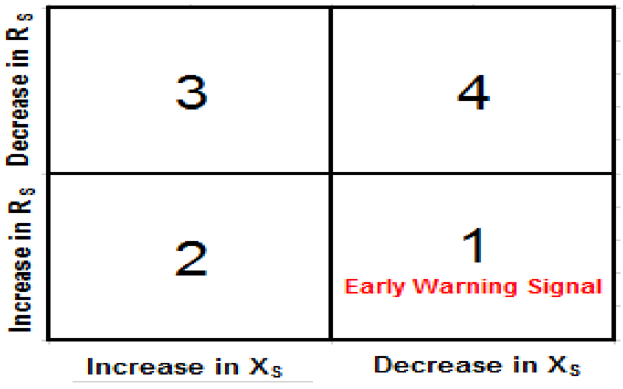


Figure 5: Scatter diagram showing the relationship between change in X_S and R_S

ing mR chart is calculated using Equations (7) and (8). X_i denotes the i^{th} measurement. There is no LCL in the mR chart because the absolute difference between consecutive values is used. If there are n consecutive measurements, we have $r = n - 1$ two-point moving range values.

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i \quad (2)$$

$$\bar{mR} = \frac{1}{r}\sum_{i=1}^{r} mR_i, \quad \text{where} \quad mR_i = |X_{i+1} - X_i| \quad (3)$$

$$CL_I = \bar{X} \quad (4)$$

$$UCL_I = \bar{X} + 2.660 * \bar{mR} \quad (5)$$

$$LCL_I = \bar{X} - 2.660 * \bar{mR} \quad (6)$$

$$CL_{mR} = \bar{mR} \quad (7)$$

$$UCL_{mR} = 3.268 * \bar{mR} \quad (8)$$

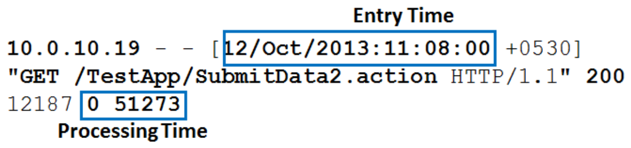
To further increase the sensitivity in eliminating noise from signals, we recommend that practitioners establish control limits in accordance with the workload profile of the system to be monitored. In our case study, we have calculated control limits for every business hour of each weekday rather than having a single control limit for all business hours of the weekday. Therefore, we have separate control limits for Monday 9:00 a.m. to 10:00 a.m., 10:00 a.m. to 11:00 a.m., 11:00 a.m. to 12:00 p.m.,.., Friday 9:00 a.m. to 10:00 a.m., 10:00 a.m. to 11:00 a.m., 11:00 a.m. to 12:00 p.m.,..,Friday 4:00 p.m. to 5:00 p.m.. We define the following rule for identifying an early-warning signal.

Early-Warning Signal Rule

X_S is below LCL and R_S is above UCL in the Individuals (X) chart for a specified number of consecutive monitoring intervals.

5. REFERENCE IMPLEMENTATION

In the real-world case study described in section 6, we used the standard web server log entries generated by the presentation tier components and application tier components. The application tier components encapsulate processing logic in the form of web services. For the data tier, we used the database snapshot. Figure 6 shows a standard log entry. The %T and %D directive is used to log the processing time taken by a request in seconds and microseconds respectively [2]. The web server log is available by default in all web applications. We conducted several tests in a controlled environment and did not see any noticeable increase in the resource utilization on enabling the processing time directive (which may requires an additional function call internally). If a web server is not available, applications can be instrumented using techniques like AOP to write similar entries in a custom log file. In [15], the overhead introduced by using AOP for data collection is reported to be almost negligible.


Entry Time
Processing Time

```
10.0.10.19 - - [12/Oct/2013:11:08:00 +0530]
"GET /TestApp/SubmitData2.action HTTP/1.1" 200
12187 0 51273
```

Figure 6: A web server log entry

We want to highlight that the proposed technique is not application specific because it is not for collecting the performance data but for discovering useful insights from the collected data. The underlying data collection mechanism can be the more efficient Application Programming Interface (API), if software system provides such data via APIs.

A health sensing script is scheduled on each component to be monitored. The script processes the newly added entries in the last S minutes and calculates the value of X_S and R_S. The calculated values are then submitted to a central collection service for storage using an API. The dashboard of the central collection service displays the Individuals control charts and generates an alert in case the Early-Warning Signal Rule defined in section 4 is violated. The calculation of X_S and R_S incurs minimal computation overhead as it requires only sum, count and a single division, making it suitable for near real-time monitoring. We have empirically determined that for the system used in our case study, setting S to two minutes provides the correct balance of acceptable performance deterioration detection capability and compute overhead. In case of a multi-tier deployment, the health sensing script is deployed on each tier and the dashboard displays one entry for each of the tier. This allows us to drill down to the tier in which the performance deterioration originated.

6. REAL-WORLD CASE STUDY

The objective of this case study is to apply the proposed technique and validate its effectiveness. The study uses X_S and R_S data from the presentation tier of the system described in section 2.

First, we use data from a randomly selected month for setting control limits. Table 1 lists the statistics of the data. The data corresponds to the working days of the selected month, containing five Mondays, five Tuesdays, four Wednesdays, four Thursdays and four Fridays. Thus, Monday and Tuesday each have 1,200 observations whereas Wednesday, Thursday and Friday each have 960 observations. During this month, no incidents of slowness were reported by the end users.

Table 1: Statistics of data used for setting control limits

Statistics	**Values**
No of days	22
No of business hours per day	8
Monitoring interval (S)	120 secs
No of observations per hour	30
No of observations per day	240
Total no of observations	5,280

We use two Individuals charts, one for throughput and second for average response time. We determine the UCL and LCL for both the charts using Equations (4) to (6). The control limits of each chart are separately calculated for every weekday and business hour combination, resulting in 40 distinct control limits. Figures 7a and 7b show the control charts for one weekday.

Next, to evaluate the accuracy of our technique in identifying early-warning signals, we used our technique on data from another 23 days. During these days, four established incidents of slowness were documented. Table 2 lists the statistics of the data used in our validation. The data corresponds to five Mondays, four Tuesdays, four Wednesdays, five Thursdays and five Fridays. Thus, Thursday and Friday each have 1,200 observations whereas Tuesday and Wednesday each have 960 observations. Monday has 1,170 observations, as one Monday had only 210 observations. This is our test dataset.

Table 2: Statistics of data used in validation

Statistics	**Values**
No of days	23
Monitoring interval (S)	120 secs
Total no of observations	5,490
Total no of slowness incidents	4

To compare our technique with those used by practitioners, we also calculated the number of pairs of X_S and R_S in the second month which satisfied the following conditions.

- $X_S < X_{S-1}$ and $R_S > R_{S-1}$
- X_S has decreased and R_S has increased by a specified value ϵ
- Using the 3-sigma control limits in Early-Warning Signal Rule, calculated using Equations (11) and (12). The symbol σ denotes the standard deviation.

$$UCL = \bar{X} + 3\sigma \tag{9}$$

$$LCL = \bar{X} - 3\sigma \tag{10}$$

- X_S is less than the LCL of the Individuals chart.
- R_S is greater than the UCL of the Individuals chart.

Table 3 summarizes the results. The count of observations in rows 4 to 6 is less than 5,490 because for every 30 data values, there are 29 two point moving range values. The number of consecutive violations was set at 1 and 2. The value of ϵ was set at 0.05 and 0.10. We use precision and recall to evaluate the accuracy of our technique. If the identified signal is not substantiated by slowness reported by end users or analysis done by an expert, the signal is considered as a false signal. If an incidence of acknowledged slowness is not detected, then it is considered as a missed signal. The precision is calculated using Equation (11) and recall is calculated using Equation (12).

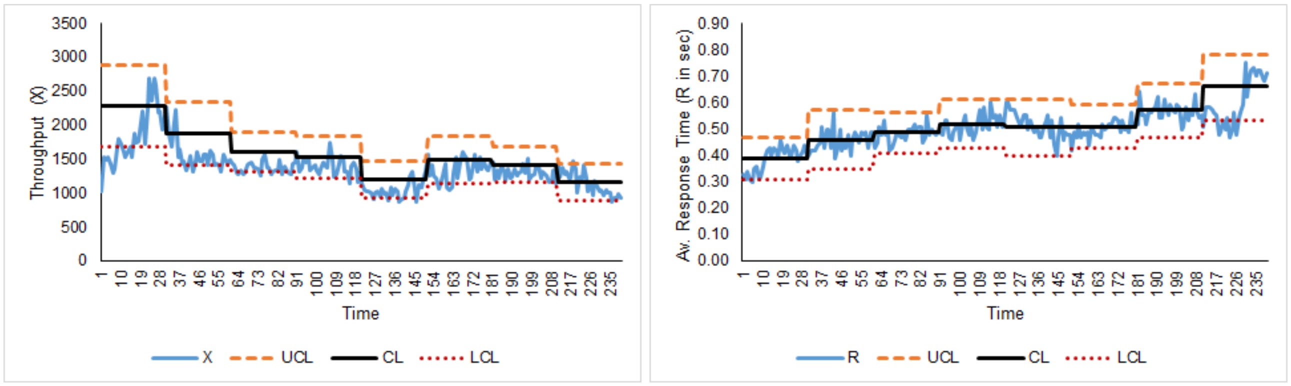
$$Precision = \frac{No\,of\,true\,signals}{No\,of\,signals\,detected\,by\,our\,technique} \tag{11}$$

$$Recall = \frac{No\,of\,true\,signals\,detected\,by\,our\,technique}{Actual\,no\,of\,early\,warning\,signals} \tag{12}$$

The results clearly confirm that our technique shows significant improvement (by reducing false alerts) than other techniques used by practitioners like independent monitoring of response time and/or throughput, using 3-sigma limits or exceeding a specified rate of change. Our technique has a recall of 100% (calculated using Equation (12)) i.e. we were able to spot all the four incidents. Out of the 32 early-warning signals our technique detected when the number of consecutive violations was set at 1, the number of signals corresponding to the four incidents was 21. Our technique has a precision of $\frac{21}{32}$ =65.63%. The precision improved significantly to $\frac{21}{21}$ =100% when the number of consecutive violations was taken as 2. In comparison, using 3-sigma limits we missed detecting any of the 21 signals. To improve alerting precision, we can tune the number of consecutive violations parameter. We conclude that our technique has helped in fulfilling the objective stated in section 2.

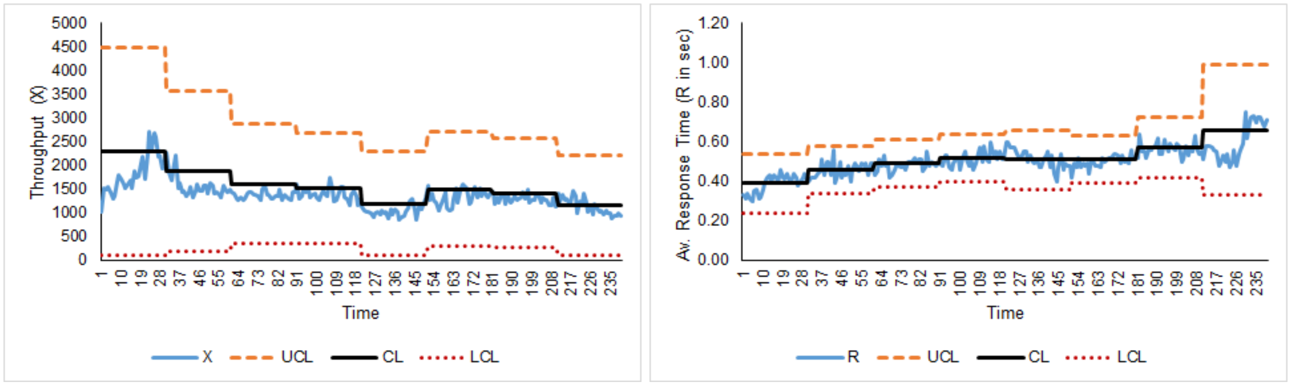
The reason for the commonly used 3-sigma thresholds, not performing well can be understood by comparing the value of σ and $\bar{mR}$. If σ is greater than $\bar{mR}$, then the tolerance limits established using Equations (11) and (12) are wider than tolerance limits established using Equations (5) and (6), leading to early-warning signals not being detected. For $\bar{X} = 1,934, \sigma = 563$ and $\bar{mR} = 183$, the 3-sigma UCL and LCL are 3,624 and 244 respectively. The UCL and LCL for Individuals chart are 2,420 to 1,447 respectively. Figure 8 shows the 3-sigma limits based control charts for the same weekday used in the Individuals charts of Figure 7.

For using control charts, it is not necessary to assume a specific distribution for the data. Tchebycheff's inequality states that for a stable process, at least $1 - \frac{1}{k^2}$ of the observations lie within k standard deviations of the mean of the observations [10]. Tchebycheff's inequality is applicable for any distribution. If k = 3, at least 88.9% of the observations, in the long duration, will lie within 3 standard deviations of the mean of the observations. The probability of false alarms in this case is 0.111. If observations are normally distributed, at least 99.73% of the observations, in the long duration, will lie within 3 standard deviations of the mean of the observations. In this case, the probability of false alarms is only 0.0027.



(a) Individuals chart for throughput (b) Individuals chart for average response time

Figure 7: XmR Chart



(a) 3-sigma control limits for throughput (b) 3-sigma control limits for average response time

Figure 8: Control Chart with 3-sigma limits

Table 3: Summary of results

Technique	**Warning Signals Detected**	**Normal Signals Detected**	**Total**
Individuals chart (consecutive violations=1)	32 (0.58%)	5,458 (99.42%)	5,490
Individuals chart (consecutive violations=2)	21 (0.38%)	5,469 (99.62%)	5,490
3-sigma control limits	0 (0%)	5,490 (100%)	5,490
X_S Down, R_S Up	1,572 (29.62%)	3,735 (70.38%)	5,307
X_S Down, R_S Up, $\epsilon = 0.05$	639 (12.04%)	4,668 (87.96%)	5,307
X_S Down, R_S Up, $\epsilon = 0.10$	214 (4.03%)	5,093 (95.97%)	5,307
X_S < LCL (of Individuals chart)	592 (10.78%)	4,898 (89.22%)	5,490
R_S > UCL (of Individuals chart)	156 (2.84%)	5,334 (97.16%)	5,490

7. THREATS TO VALIDITY

Internal Validity: The proposed technique uses data from an software system used in industry. The data is historical and may be influenced by the version of the software system corresponding to the time of data. The data reflects the real-world behaviour as it is also not possible to impose any kind of controls in a real-world system and may influence the accuracy of our findings. To reduce this threat, we have used performance data corresponding to one complete month.

External Validity: In the paper, we validate our approach using a mission critical software system from industry. To generalize our technique further, additional software systems from industry and publicly available e-commerce benchmarking applications like DellDVD [1] can be included. The monitoring interval used in this paper has been empirically determined and found to work well for the application used in the case study. The same cannot be generalized for all applications without more tests.

We plan to address the above limitations as an extension of this work in future.

8. CONCLUSION

The technique proposed by us has helped in institutionalizing a structured and systematic approach for system monitoring teams to proactively spot early-warning signs of performance related issues using two application metrics and take corrective action. This is a significant improvement over the existing system monitoring methods which is mostly silo-based in which different dimensions of IT infrastructure like server compute, storage, network, middleware and database are monitored in isolation and not linked to application service metrics like response time and throughput. It is often seen that software projects in industry acquire complex system monitoring tools but are unable to use them effectively in their daily operations because of the complexity and abundance of performance counters being monitored, level of expertise needed in setting realistic thresholds. The advantage of the proposed technique is its simplicity, intuitiveness, low implementation overhead and accuracy. The technique and its underlying concept complements and strengthens existing system monitoring tools typically used by practitioners in industry.

Our future work will include experiments on open source benchmark applications to further validate our model. We also intend to do a comprehensive study on the anomaly detection and alerting techniques provided by commercial and open source system monitoring tools like SCOM [4], OEM [6] and Nagios [5].

9. ACKNOWLEDGMENTS

We wish to thank Nidhi Dahiya, Vikash Vardhan, Gopal Sharma, Madhuri Kumari and other anonymous reviewers for their valuable feedback on improving this work.

10. REFERENCES

[1] Dell DVD Store Database Test Suite. http://linux.dell.com/dvdstore/. Online; accessed 21 February 2017.

[2] Logging control in w3c httpd. https://www.w3.org/Daemon/User/Config/Logging.htmlcommon-logfile-format. Online; accessed 25 September 2016.

[3] Microsoft System Center: Monitors and Rules. https://technet.microsoft.com/en-us/library/hh457603(v=sc.12).aspx. Online; accessed 23 February 2017.

[4] Microsoft System Center Operations Manager. https://technet.microsoft.com/library/hh205987.aspx. Online; accessed 21 February 2017.

[5] Nagios - The Industry Standard In IT Infrastructure Monitoring. https://www.nagios.org/. Online; accessed 21 February 2017.

[6] Oracle Enterprise Manager 12c. http://www.oracle.com/technetwork/oem/enterprise-manager/overview/index.html. Online; accessed 21 February 2017.

[7] F. M. Bereznay. Did something change? using statistical techniques to interpret service and resource metrics. In *Int. CMG Conference*, 2006.

[8] J. P. Buzen and A. W. Shum. Masf - multivariate adaptive statistical filtering. In *Int. CMG Conference*, pages 1–10. Computer Measurement Group, 1995.

[9] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 452–461, June 2008.

[10] W. A. Florac and A. D. Carleton. *Measuring the Software Process: Statistical Process Control for Software Process Improvement.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *2010 10th International Conference on Quality Software*, pages 32–41, July 2010.

[12] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action.* Cambridge University Press, New York, NY, USA, 1st edition, 2013.

[13] S. Iwata and K. Kono. Narrowing down possible causes of performance anomaly in web applications. In *2010 European Dependable Computing Conference*, pages 185–190, April 2010.

[14] J. D. C. Little. A proof for the queuing formula: L = λw. *Oper. Res.*, 9(3):383–387, June 1961.

[15] J. P. Magalhaes and L. M. Silva. Detection of performance anomalies in web-based applications. In *2010 Ninth IEEE International Symposium on Network Computing and Applications*, pages 60–67, July 2010.

[16] H. Malik, B. Adams, and A. E. Hassan. Pinpointing the subsystems responsible for the performance deviations in a load test. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 201–210, Nov 2010.

[17] H. Malik, B. Adams, A. E. Hassan, P. Flora, and G. Hamann. Using load tests to automatically compare the subsystems of a large enterprise system. In *2010 IEEE 34th Annual Computer Software and Applications Conference*, pages 117–126, July 2010.

[18] H. Malik, H. Hemmati, and A. E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1012–1021, May 2013.

[19] H. Malik, Z. M. Jiang, B. Adams, A. E. Hassan, P. Flora, and G. Hamann. Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 222–231, March 2010.

[20] R. K. Mansharamani, A. Khanapurkar, B. Mathew, and R. Subramanyan. Performance testing: Far from steady state. In *COMPSAC Workshops*, pages 341–346. IEEE Computer Society, 2010.

[21] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 299–310, New York, NY, USA, 2012. ACM.

[22] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Conference on Networked Systems Design &*

Implementation - Volume 3, NSDI'06, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[23] L. Tang, T. Li, L. Shwartz, F. Pinel, and G. Y. Grabarnik. An integrated framework for optimizing automatic monitoring systems in large it infrastructures. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1249–1257, New York, NY, USA, 2013. ACM.

[24] I. Trubin. Review of IT Control Chart. *Journal of Emerging Trends in Computing and Information Sciences*, 4(11):857–868, Dec. 2013.

[25] I. Trubin and V. C. Scmg. Capturing workload pathology by statistical exception detection system. In *Proceedings of the Computer Measurement Group*, 2005.

[26] I. A. Trubin. Global and Application Level Exception Detection System, Based on MASF Technique. In *28th International Computer Measurement Group Conference, December 8-13, 2002, Reno, Nevada, USA, Proceedings*, pages 557–566, 2002.

[27] I. A. Trubin and L. Merritt. "Mainframe Global and Workload Level Statistical Exception Detection System, Based on MASF". In *30th International Computer Measurement Group Conference, December 5-10, 2004, Las Vegas, Nevada, USA, Proceedings*, pages 671–678, 2004.

[28] T. Wilson. What were they thinking: Modeling think times for performance testing. 2011.