IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-7, Issue-10, Oct 2019 E-ISSN: 2347-2693

Reference Model for Effective Performance and Availability Monitoring in Large Scale Software Systems

Raghu Ramakrishnan[1*], Arvinder Kaur[2]

[1]Tata Consultancy Services, C-56, Phase II, Noida, Uttar Pradesh, India
[2]USICT, Guru Gobind Singh Indraprastha University, Delhi, India

[*]*Corresponding Author: raghuramakrishnan71@gmail.com*

DOI: https://doi.org/10.26438/ijcse/v7i10.9097 | Available online at: www.ijcseonline.org

Accepted: 11/Oct/2019, Published: 31/Oct/2019

***Abstract*—** The monitoring of different parts of the software stack is essential for ensuring acceptable performance and availability of large scale heterogeneous software systems. However, given that a large amount of data is generated by various parts of the software stack, identifying the minimum set of data elements for inclusion under the initial monitoring umbrella is challenging. Although the elements are similar across the majority of the projects, we have seen that teams often spend considerable effort and time in identifying them. In this paper, we present a layered monitoring reference model, with different layers targeting different parts of the software stack using appropriate data elements. The model provides guidance on the minimum set of data elements, drawing on learnings from more than 20 real-life projects. The model also explains the data elements and the motivation for including them in the model.

***Keywords*—** System monitoring, Production systems, Performance monitoring, Availability monitoring

I. INTRODUCTION

Large-scale software systems are ubiquitous in various business domains. The characteristics of such systems are a large number of users and transactions, complex processing needs, multi-tier deployment, and several custom-built, OS (Open Source) or COTS (Commercial off-the-shelf) components. The systems also interface with various 3rd party systems. Besides, the systems evolve continuously over time and must adhere to stringent performance and availability requirements.

The interaction between the different components that make up the system influences the runtime behavior of a large-scale software system. During the Operations and Maintenance (O & M) phase, the production support team continuously monitors the vitals of the system to gain insights into the availability and performance of the system. The heterogeneous nature and complexity of large-scale software systems make monitoring difficult. Any downtime or slowness may result in significant monetary losses, a negative impact on the brand image of organizations, and loss of productivity. Apart from ensuring system uptime and performance, proactive monitoring can also reduce operating costs and improve the quality of services organizations provided to their customers.

For monitoring the system behavior in production, the technical operations team must set up the initial version of monitoring practice before transitioning from the development phase to the O & M phase of the development lifecycle. System monitoring tools, as they are popularly called in the industry, have found their place in organizations for monitoring of large-scale software systems. However, given a large amount of monitoring data generated by various parts of the software stack, it is challenging to identify the minimum set of data elements that one needs to include under the initial monitoring umbrella. We refer to this set as the Minimum Monitoring Data Elements (MMDE). Although the data elements to be collected are usually similar across the majority of the projects, we have seen that teams often spend considerable effort and time in the process of identifying them. Further, there needs to be a trade-off between the type of data in the MMDE and performance overhead introduced in collecting them. To address the above issue, we propose a reference model that will act as a springboard for setting up the initial version of monitoring for software systems transitioning to the O & M phase. The proposed model is a framework comprising of guidelines summarized from more than 20 real-life software systems from the public sector domain. The model is agnostic of the tool implemented for monitoring. The model derived from practical projects provides a perspective of hands-on users and may be valuable for both academic researchers and industry practitioners.

The rest of the paper is organized as follows, Section I contains the introduction. Section II reviews the related work. Section III describes the proposed reference model. Section IV lists the considerations related to analyzing the monitoring data collected by the model, and Section VI concludes this paper and provides directions for future work.

II. RELATED WORK

The current work in monitoring the performance and health of software systems covers the data elements to be measured, analysis techniques, and tools. The data collected as part of monitoring include CPU utilization, memory utilization, disk, and network utilization, processing, or response times, with most of the data being time-series [1][2][3][4]. Researchers have proposed several anomaly detection techniques in software performance engineering literature.

Acharya et al. propose PerfAnalyzer, a proactive performance monitoring framework for building health models for detecting performance degradation in a production environment and validate the same on five performance counters: CPU, memory, I/O, disk, and network [5]. Control charts have been used to monitor and detect anomalies in performance counters collected during operations [1][6][7]. Silva et al. use aspect-oriented programming and system monitoring tools for collecting performance data and process it using data correlation and time-series alignment methods for detecting performance anomalies [8]. Malik et al. describe a supervised learning technique called WRAPPER to reduce the number of performance counters collected during monitoring of low-level infrastructure and compare its effectiveness with unsupervised learning techniques like random selection, clustering, and PCA (Principal Component Analysis) [4]. Cherkasova et al. first characterize the behavior of transactions using characterizations like application performance signatures and regression-based transaction models and then use the characterizations to detect performance anomalies [9][10]. Jiang et al. summarize the various techniques used for analyzing load testing results like verifying against thresholds, detecting unusual behavior, and known types of problems [11]. Bereznay explores and compares the use of statistical techniques like Hypothesis Testing, Statistical Process Control (SPC), Multi-variate Adaptive Statistical Filtering (MASF), and Analysis of Variance (ANOVA) for detecting a deviation in metrics [12]. Apart from the above work, there are other methods like martingale and deep learning for detecting anomalies in time series data, which easily apply to performance data [13][14].
There are several well-established OS and COTS monitoring tools in the Enterprise Monitoring System (EMS) and Application Performance Management (APM) domain. Zabbix[1], Nagios[2], and Prometheus[3] are in the OS space, whereas the COTS space includes tools like Oracle Enterprise Monitoring (OEM)[4], CA Infrastructure Management[5], Dynatrace[6], AppDynamics[7]. Cloud service providers are also offering monitoring and management services like Microsoft Azure Monitor[8], Amazon CloudWatch[9], and Google Stackdriver[10] for getting complete insights into cloud resources and applications hosted on them.

III. PROPOSED REFERENCE MODEL

In this section, we describe our reference model based on a layered monitoring architecture with different layers targeting different parts of the software stack. The proposed model is a conceptual framework and agnostic of the actual monitoring tool in use. The model is based on the learnings from more than 20 real-life software systems having the following characteristics.

- The systems are business-critical with a large number of users and transactions.
- Most of the systems have a public-facing portal or website on the internet and a back-office portal accessed by department users on an intranet. The internet portal is accessible 24 x 7 x 365.
- The systems experience peak periods when deadlines like the quarter ending and financial year ending. For instance, statutory filings like income tax returns and company reports, tax assessments, government treasury operations are systems that witness peak periods.
- The systems use components like web servers, application servers, message queues, databases, and packaged applications, which are either OS or COTS.
- The systems run on virtual machines (VMs) or bare-metal servers.
- The systems communicate with various other systems using web services, APIs, and managed FTP.
- The tools used for monitoring availability and performance are custom-developed, OS, and COTS.

The different parts of the software stack form a layered architecture that includes low-level infrastructure, communication and processing middleware, application, and

[1]https://www.zabbix.com/
[2]https://www.nagios.org/
[3]https://prometheus.io/
[4]https://www.oracle.com/technetwork/oem/sys-mgmt/index.html
[5]https://www.ca.com/us/products/application-and-infrastructure-monitoring.html
[6]https://www.dynatrace.com/
[7]https://www.appdynamics.com/
[8]https://azure.microsoft.com/en-in/services/monitor/
[9]https://aws.amazon.com/cloudwatch/
[10]https://cloud.google.com/stackdriver/

end-user. In a layered architecture, each layer provides service to the layer above it and receives service from the underlying layer. Any issue in one of the layers will have a domino effect on all the layers above it.

Based on our detailed analysis of the monitoring practices and mechanisms in the real-world software systems, we found that it is possible to arrange the monitoring components of software systems in four distinct groups of related functions or layers. Fig. 1 shows the monitoring layers corresponding to the equivalent layers of the software stack. The monitoring components collect data in the form of metrics (e.g., processor and memory utilization percentage), logs (e.g., web server log), and snapshots (e.g., storage, database). Metrics are numbers that describe some behavior of the component at a given point in time, whereas logs contain raw data emitted by the component. The snapshots capture several metrics associated with one component at a point in time. The snapshots are typically collected less frequently than metrics because there may be an overhead associated while collecting them. All the examples used in this paper are from real-world systems.

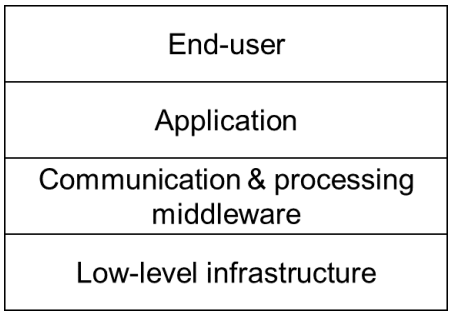

Figure 1. Layered monitoring architecture

- **Low-level infrastructure**: The tier comprises virtual machines and bare-metal servers with computing, memory, storage, network, and security devices like links, load balancers, and firewalls.
- **Communication and processing middleware**: The tier comprises of supporting software components like web and application servers, message queues, API (Application Processing Interface) Gateway, Document Management System (DMS), Business Process Management (BPM) Identity and Access Management (IAM), Relational Database Management Systems (RDBMS) and No-SQL databases.
- **Application**: The tier includes bespoke or custom-developed applications like web applications, APIs, batches, and packaged software like Customer Relationship Management (CRM), Enterprise Resource Planning (ERP).
- **End-user:** The tier is the user touchpoint for the software system like browsers and mobile devices.

We now describe the data elements collected by the monitoring entities of the different layers and the motivation for including them.

A. Low-level infrastructure

Guideline A1 (METRIC): For compute and memory, monitor processor utilization percentage, memory utilization percentage, and swap utilization percentage.

Motivation: We have observed that applications experience performance issues when they are stranded for resources. High processor utilization may indicate a need for reducing the compute footprint by tuning the application logic, database queries, or Java Virtual Machine (JVM). Swap utilization helps assess if there is a paging issue due to a shortage of memory.

Guideline A2 (METRIC): For local storage, monitor the NFS (Network File System) and SAN (Storage Area Networks) measure percentage space utilized by directories of the software system.

Motivation: We have observed that low or out of space situations often cause system slowness or unavailability. For example, if the directories associated with DMS becomes full, no new documents can be added, and calls may block resulting in users experiencing slowness due to hung threads.

Guideline A3 (METRIC): For LUNs (Logical Unit Number), monitor utilization percentage, IOPs (Input/Output Operations Per Second) or throughput, queue length, and response time or latency. The response time includes time spent in waiting for getting service and time spent in getting service. Fig. 3 displays an extract of a disk utilization graph with the utilization ranging from 85% to 95% from 13:05 and 14:10.

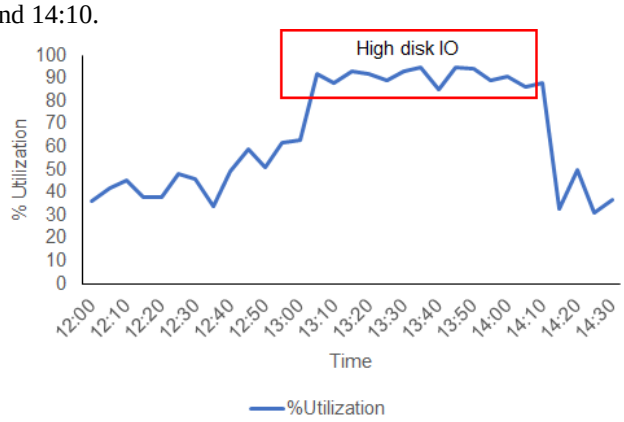

Figure 2. Disk utilization showing high IO

Motivation: We have seen that slowness occurs whenever there is a continuous high IO rate or observed IOPs approaches the theoretical IOPs capacity of the associated LUNs (Logical Units). LUNs associated with logs, database, and DMS are more prone. Looking at throughput and utilization percentage can help identify bottleneck LUNs.

The higher the queue length for a LUN, the higher is the response time. Nowadays, deployments may use tiered storage pools comprising of fast SSD (Solid State Drive) and slow NL-SAS (Near Line Serial-Attached SCSI) disks. In some cases where there is tiered storage, we have noticed instances of system slowness caused by the spillover effect in which a portion of frequently accessed data has spilled over to the slower disks after the space in the faster disks was exhausted.

Guideline A4 (METRIC): For network, monitor the utilization percentage of both uplink and downlink.
Motivation: Network links connect (a) the Data Center (DC) with the internet or intranet cloud and (b) DC with Disaster Recovery site (DR). The DC to DR link replicates data between both locations and in some cases, for accessing services hosted on the secondary site. The commonly used replication strategies include storage level, database level, and custom-built. High utilization of the links on a sustained basis indicates that the link capacity might be below the actual demand, which can result in the degradation of application performance and user experience. Similarly, high utilization of the link used for replicating data between DC and DR can result in application slowness if the replication mode is synchronous. However, the probability of data losses in the event of DC failure increases if the replication mode is asynchronous.

Guideline A5 (METRIC): Monitor the CPU utilization of the firewall. A firewall is usually the entry point for end-user traffic.
Motivation: The high CPU usage affects incoming traffic resulting in end-users complaining of slowness. We have observed that inspection of content in the incoming traffic against a growing library of signatures is the usual cause for the high CPU use.

Guideline A6 (METRIC): Monitor the memory of the load balancers or application delivery controllers (ADC). ADC is a device placed between the firewall and one or more web or application servers. The device performs functions like load balancing, application acceleration, SSL offloading, rate shaping, and WAF (web application firewall). Load balancers or ADCs form the second touchpoint for end-user traffic after the firewall.
Motivation: The high memory usage can throttle incoming traffic, causing end-users to complain of system slowness. We have observed that one of the leading causes of high memory utilization is the use of advanced cipher suites by current browsers.

Guideline A7 (METRIC): Use ping at predefined intervals for monitoring the availability of the different low-level infrastructure components like servers, load balancers, and routers. Ping may be disabled from outside the DC but is usually allowed within the DC.
Motivation: The use of periodic availability checks helps detect if any of the low-level infrastructure components are down or not responding. If a server is not responding, the communication and processing middleware and application deployed on the server may also not function correctly.

B. Communication and processing middleware

Guideline B1 (LOG): For web servers, check for the MaxClients warning or error message in the Http error log. The number of requests with Http codes other than 200 (success) and 302 (redirect) needs to be collected. Fig. 3 shows the MaxClients error message seen in the error log of a web server.

```
[error] server reached MaxClients setting, consider raising the MaxClients setting
```

Figure 3. MaxClients error message in the web server log

Motivation: If the MaxClients value is reached, then either the webserver is receiving a higher number of requests than it is configured to handle, or the backend is responding slowly, resulting in many web threads becoming busy. In the latter case, the real cause may be garbage collection cycles freezing the application server or an expensive query executing in the database. The system performance will suffer because incoming requests will start getting queued up the host operating system.

Guideline B2 (METRIC): For application servers, monitor counters related to web container thread pool like the number of concurrently active threads, percentage of the pool that is in use, and percentage of the time all threads are in use. Fig. 4 shows 8-13 concurrently active threads in thread pool configured with 30 threads.

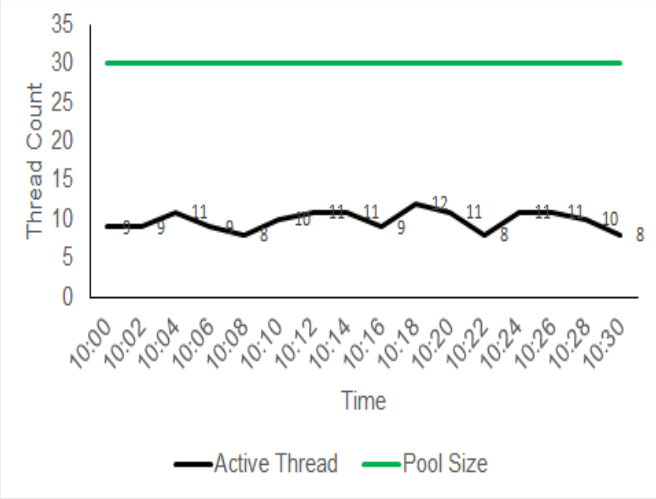

Figure 4. Web container thread pool showing active threads and pool size

Motivation: A consistently high percentage of the pool that is in use or number of concurrently active threads indicates the application server is receiving a higher number of requests than it is configured to handle. Also, the application logic executed by the threads may be slow, resulting in threads becoming busy. The web server threads which originated the call to the application server will also become busy, resulting in users experiencing slowness.

Guideline B3 (METRIC): For application servers, monitor counters associated with the database connection pool. The counter includes the number of free connections, threads waiting for a connection to become available, percentage of the pool in use, percentage of the time all connections are in use, average use time, and average wait time.
Motivation: If the percentage of used connections or the number of threads waiting for a connection to become available is very high, it may point to the connection pool is not sized correctly or the database queries running slowly and requiring tuning. If the count of free connections drops to zero and does not increase, there may be a possibility of a connection leak.

Guideline B4 (SNAPSHOT): For assessing the health of the JVM, the percentage of time spent in processing real transactions vs. time spent in garbage collection (GC) activity. Also, watch the average interval between stop the world (STW) GC events, and average/maximum latency experienced during GC event pauses. Fig. 5 shows garbage collection activity from one of the JVM of a software system. We observe that the GC time varies between 86ms to 240ms, and the time between to GC events is 3s to 34s. The overall time spent on GC is 1.796s in the monitoring interval of 149s, resulting in an overhead of 1.2%. Fig. 6 shows the JVM memory before and after the garbage collection cycle. For instance, after the first cycle, the free memory increased from 0 to 425 MB.

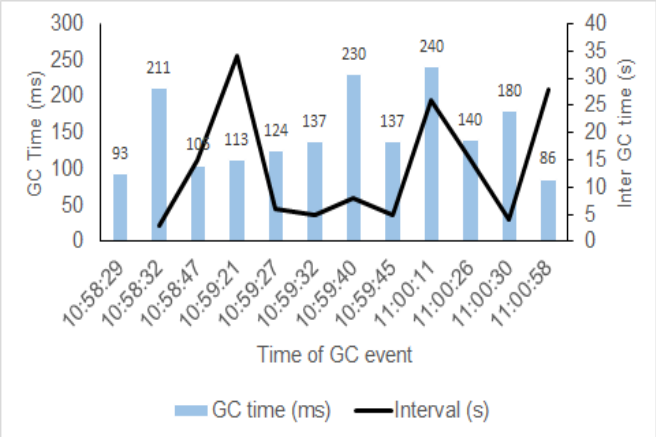

Figure 5. GC graph of a JVM showing GC and inter-GC time

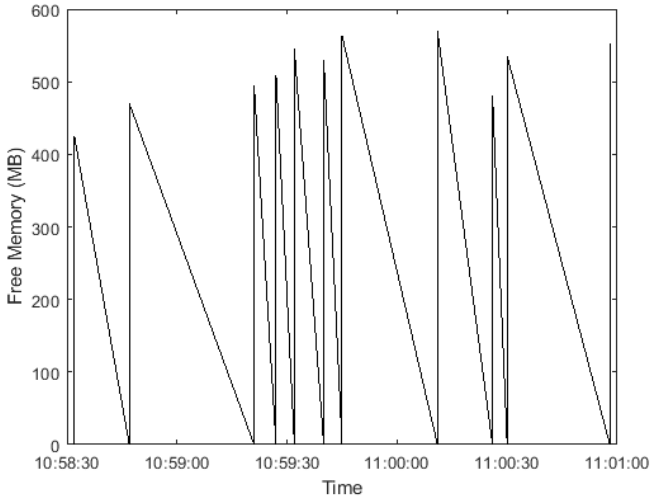

Figure 6. GC graph of a JVM showing shortlived objects (nursery) memory before and after GC

Motivation: If the latency of STW GC events is high, or the interval between them is short, requests to the applications deployed on the JVM will experience slowness or appear stalled. In a domino effect, this will increase the number of used web container threads and in turn, database connections. High GC rate and pauses may be indicative of high object creation rates or small young generation space resulting in objects prematurely moving to old generation space.

Guideline B5 (METRIC): The use of message queues is typical in large-scale software systems for asynchronously exchanging messages. For message queues, include the queue length, waiting, and service time in the queue in the monitoring.
Motivation: If the waiting time in the queue shows an increase, or there is a build-up of entries in the queue, the applications may not be consuming the messages fast enough. The build-up may impact the overall user experience directly or indirectly.

Guideline B6 (LOG): Implement keyword-based monitoring to flag events of interest like stuck or hung web container threads and out of memory errors. Fig. 7 shows a hanging thread warning, which was found by searching for "WSVR0605W" and "may be hung" keyword in the IBM Websphere application server log.

```
ThreadMonitor W   WSVR0605W: Thread "Default : 0"
(<ThreadID>) has been active for xxx milliseconds and
may be hung.  There is/are 1 thread(s) in total in the
server that may be hung.
```

Figure 7. Hung thread warning in the application server log

Motivation: The use of keyword-based search is a simple but powerful technique for detecting errors and early warning signals. If left undetected, the condition may eventually

impact the performance and availability of the software system. The above example is an early warning signal because if there is a build-up of hung threads, the web container thread pool may run out of threads, resulting in performance degradation as new requests start getting buffered by the host operating system.

Guideline B7 (SNAPSHOT): For database servers, monitor at least the ratio of reads from the buffer pool to the reads from storage, average CPU time per query, average IO time per query, average elapsed time per query. Statistics like the number of executions of a query, average rows read need to be reported and analyzed regularly. Table 1 shows a tabular extract of the Oracle AWR (Automatic Workload Repository) report generated from the database of a system. In the first query, the database spent 4.16% of 25,519.55 s on computing, and 95.9% on doing IO.

Table 1. Extract of an AWR report from an Oracle database instance

Elapsed Time (s)	Executions	Elapsed time per exec (s)	% Total	% CPU	% IO	SQL Text
25,519.55	51	500.38	18.31	4.16	95.9	SEL.
6,022.47	12	501.87	4.33	1.28	94.1	SEL.

Motivation: Our experience shows that expensive queries are one of the most identified reasons for application unavailability and poor performance. There may be a need to tune the query or add appropriate indexes if the average CPU or IO time of a query is high. The query may be waiting on a lock if there is a significant difference between the CPU time and the execution time. It may be possible to reduce the number of query executions using techniques like removing redundant calls or caching the results. A high number of average rows read are high may be symptoms of a full table scan or a generic filter criterion in the query. The overall objective is to minimize the total time i.e., CPU, IO, and others.

Guideline B8 (METRIC): For monitoring availability of the communication and processing middleware, use a mix of strategies like periodic checking of a test page for web and application server, querying a table in the database, or executing telnet to the listening port. It is recommended to validate the output using regular expressions or fixed values if using a test page or database table query.

Motivation: The checking of availability helps detect if any of the communication and processing components are down or not responding. If a component is not responding, the deployed application may also not function correctly, impacting the end-user experience.

C. Application

Guideline C1 (METRIC): For monitoring application performance, measure the average/median response time, and throughput related metrics for the business services.

Motivation: The average response time and average throughput in systems are inversely related. If there is an increase in the average response time with a corresponding decrease in throughput, it may indicate a performance anomaly that needs remediation.

Guideline C2 (METRIC): Configure the monitoring of availability and performance of the application by accessing select pages (e.g., the home page) at predefined time intervals using synthetic transaction monitoring and checking the response or success using fixed values or regular expressions. Checking for only an Http status 200 may result in an error condition not being caught.

Motivation: The adoption of synthetic transaction monitoring helps in detecting failures before they get reported from the field. The failures may have originated in any of the underlying layers. Monitor the performance and availability of applications, even if real users are not currently accessing them.

Guideline C3 (METRIC): The execution time, pass or fail status, and the number of records processed is to be measured for batches. The concurrent number of batches running in a given time also needs to be included.

Motivation: In large-scale software systems, batches are used to carry out internal activities or processing data received from or generating data for external systems. Examples include a reconciliation batch run at the end of the day to reconcile credit card payments or bulk submission of application forms by a third-party. Batch failures may impact the level of business services rendered to users.

Guideline C4 (LOG): We also recommend monitoring of application-specific errors using keyword-based searching in the application logs. Fig. 8 shows a directory creation error detected by searching for "cannot create directory" and "java.io.IOException" keyword in the application log of a software system.

```
java.io.IOException: Cannot create directory
/xxxx/yyyy/zzzz at
java.lang.Throwable.<init>(Throwable.java:67)
```

Figure 8. java.io.IOException in the application log

Motivation: As explained under B6, keyword-based search helps in the timely detection of errors, which, if left undetected, may impact the performance and availability of the system.

D. End-user

Guideline D1 (METRIC): Implement real user measurement of traditional metrics like Onload time, Time to

First Byte (TTFB), and newer metrics like First Contentful Paint (FCP) and Time to Interactive (TTI). Onload time, TTFB, and FCP with custom wrappers built on the underlying native navigation and paint timing supported by the latest browsers. To measure TTI for real users, we may need to use a COTS tool. Dheeraj et al. include performance as one of the four measures for quantifying software product quality [15]. Arush et al. use tools like Lighthouse, WebPageTest, and PageSpeed Insights to measure end-user metrics [16].

Motivation: Real user monitoring helps capture key user experience metrics, aggregated from actual users in the field.

IV. DISCUSSION

We have found that it is possible to detect most performance and availability issues using MMDE and take quick corrective action. For detailed root-cause analysis, more details may need to be collected. In this section, we explain considerations that need addressing even after obtaining the minimum subset of monitoring data.

The frequency of data collection is an important point. If the sampling is frequent, it may introduce performance overhead. If the sampling frequency is very less, significant events of interest with may point to performance or availability issues may not be detected. There must be a trade-off between the rate of sampling and the possibility of misses.

The data collected from a monitoring layer needs to be analyzed for identifying unusual behavior. The data from various monitoring layers is also to be correlated to point the layer and component where the anomaly may have originated. For instance, an increase in response time in the end-user layer may correlate with high CPU utilization in the low-level infrastructure layer during some interval. It is relatively simple to identify anomalies like unavailability of a part of the software stack, presence of warning or error messages in log files. Other anomalies, like a significant change in response time or throughput over time, are more challenging to detect. OS and COTS monitoring tools like Zabbix [11], Oracle Enterprise Monitoring [12], CA Unified Infrastructure Management [13], Microsoft SCOM [14], and Dynatrace[15] provide for setting fixed or adaptive threshold values. The tools derive adaptive thresholds by applying proprietary algorithms or statistics like mean, median, 95th percentile, k standard deviations on the historical performance data. In some tools, it is possible to have different threshold values for different times of the day or week to consider changing workloads. There is no shortage of advanced techniques for anomaly detection and data correlation with one of the cloud service providers even offering anomaly detection in time-series data as a service[16]. However, the packaging of these techniques into the tools usable by the industry and characterization of which methods are useful in what scenarios still remains unaddressed. To overcome this limitation, we recommend extracting the performance data collected by these tools into a separate data repository and carrying out the analysis and alerting using that data.

The data and analysis need to be presented intuitively and interactively to help monitoring teams to focus on areas requiring immediate attention. Grafana[17] is an OS platform for building analytics and monitoring dashboards using data from various data sources like Graphite, Cloudwatch, Prometheus, InfluxDB, MySQL, PostgreSQL.

V. CONCLUSION AND FUTURE SCOPE

Monitoring is critical to ensuring the performance and availability of large-scale software systems. Determining the minimum but sufficient set of data to be collected as part of monitoring is a challenging yet essential activity for the technical operations team. Therefore, we propose a reference model, consisting of 20 guidelines compiled from more than 20 real-life software systems for addressing this challenge. The model will be beneficial for teams in setting up the initial version of monitoring for software systems transitioning to the O & M phase. We plan to extend this work in the future to include guidelines for monitoring emerging technologies like containers, API gateways. We also intend to evaluate the performance of popular anomaly detection techniques on time-series data from real-world systems.

ACKNOWLEDGMENT

We wish to thank Vikash Vardhan for providing feedback and guidance on this work.

REFERENCES

[1] T.H.D. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, P. Flora, "*Automated Detection of Performance Regressions Using Statistical Process Control Techniques*", In the Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, USA, pp. 299–310, 2012.

[2] H. M. Alghmadi, M. D. Syer, W. Shang, A. E. Hassan, "*An Automated Approach for Recommending When to Stop*

[11]https://www.zabbix.com/documentation/4.2/manual/config/triggers

[12]https://docs.oracle.com/cd/E24628_01/doc.121/e24473/adv_threshold.htm#EMADM15131

[13]https://docops.ca.com/ca-unified-infrastructure-management-probes/ga/en/how-to-articles/configuring-alarm-thresholds

[14]https://social.technet.microsoft.com/wiki/contents/articles/237.scom-how-self-tuning-threshold-baseline-is-computed.aspx

[15]https://www.dynatrace.com/platform/artificial-intelligence/anomaly-detection/

[16]https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/time-series-anomaly-detection

[17]https://grafana.com/grafana/plugins?type=datasource

Performance Tests", In the Proceedings of the IEEE International Conference on Software Maintenance and Evolution, USA, pp. 279-289, 2016.

[3] S. Ghaith, M. Wang, P. Perry, Z. M. Jiang, P. O'Sullivan, J. Murphy, "*Anomaly detection in performance regression testing by transaction profile estimation*", Journal of Software Testing, Verification and Reliability. Vol. 26, Issue. 1, pp. 4–39, 2016.

[4] H. Malik, H. Hemmati, A. E. Hassan, "*Automatic detection of performance deviations in the load testing of Large Scale Systems*", In the Proceedings of the International Conference on Software Engineering, USA, pp. 1012-1021, 2013

[5] M. Acharya, V. Kommineni, "*Mining Health Models for Performance Monitoring of Services*", In the Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, New Zealand, pp. 409–420, 2009.

[6] S. Iwata, Kono, K, "*Narrowing Down Possible Causes of Performance Anomaly in Web Applications*", In the Proceedings of Dependable Computing Conference, USA, pp. 185–190, 2010.

[7] I. Trubin, "*Capturing workload pathology by statistical exception detection systems*", In the Proceedings of International Computer Measurement Group Conference, USA, 2005.

[8] L. M. Silva, J. P. Magalhães, "*Detection of Performance Anomalies in Web-Based Applications*", In the Proceedings of IEEE International Symposium on Network Computing and Applications, USA, pp. 60-67, 2010.

[9] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, E. Smirni, "*Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change*", In the Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, USA, pp. 452-461, 2008.

[10] N. Mi, L. Cherkasova, K. Ozonat, J. Symons, E. Smirni, "*Analysis of application performance and its change via representative application signatures*", In the Proceedings of IEEE Network Operations and Management Symposium, Brazil, pp. 216–223, 2008.

[11] Z.M. Jiang, A.E. Hassan, "*A Survey on Load Testing of Large-Scale Software Systems*", IEEE Transactions on Software Engineering. Vol. 41, No. 11, pp. 1091–1118, 2015.

[12] F.M. Bereznay, "*Did something change? using statistical techniques to interpret service and resource metrics*", In the Proceedings of the International Computer Measurement Group Conference, USA, 2006.

[13] Shen-Shyang Ho, "*A martingale framework for concept change detection in time-varying data streams*", In the Proceedings of international conference on Machine learning, Germany, pp. 321-327, 2005.

[14] D. T. Shipmon, J. M. Gurevitch, P. M. Piselli, S. T. Edwards, "*Time Series Anomaly Detection; "Detection of anomalous drops with limited features and sparse examples in noisy highly periodic data*", Vol. abs/1708.03665, CoRR, 2017.

[15] Dheeraj, K. Sharma, "*Proposed 4S Quality Metrics and Automated Continuous Quality (ACQ) Metrics Dashboard to Quantify Software Product Quality*", International Journal of Computer Sciences and Engineering, Vol. 7, Issue. 1, pp.865-869, 2019.

[16] A. Agarwal, A. Dixit, "*Progressive Web Applications: Architectural Structure and Service Worker Asset Caching*", International Journal of Computer Sciences and Engineering, Vol. 7, Issue. 9, pp. 127-139, 2019.

Authors Profile

Raghu Ramakrishnan is Chief Architect and Head, Technology in Public Sector Unit of Tata Consultancy Services. He has over 25 years of experience in designing and performance engineering business critical software systems for government, airlines and financial institutions. He is pursuing research in University School of Information and Communication Technology, Guru Gobind Singh Indraprastha University, Delhi, India. He is member of the IEEE and Computer Society of India..

Arvinder Kaur is Professor in University School of Information and Communication Technology, Guru Gobind Singh Indraprastha University, Delhi, India. Her research interests include Software Engineering, Software Testing, Software Metrics, Fault Prediction and Project Management. She has authored more than 80 research papers in International Journals and Conferences.

