iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 9 **Issue:** XII **Month of publication:** December 2021

DOI: https://doi.org/10.22214/ijraset.2021.39672

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 9 Issue XII Dec 2021- Available at www.ijraset.com

A Study on Framework for Anti-Pattern Evaluation of System Software

Shaik Mastanvali[1], Dr. Neeraj Sharma[2]
[1]*Research scholar at CSE, Department, SSSUTMS-Sehore, Bhopal*
[2]*Associate Professor, Department of CSE, SSSUTMS-Sehore, Bhopal*

Abstract: ***Studies with a variety of viewpoints, goals, measurements, and quality characteristics have been conducted in order to determine the effect of design patterns on quality attributes. This has resulted in findings that are contradictory and difficult to compare. They want to explain these findings by taking into account confounding variables, practises, measurements, and implementation problems that have an impact on quality. Furthermore, there is a paucity of research that establishes a link between design pattern assessments and pattern creation studies, which is a significant limitation. For the purpose of detecting and categorising software performance anti-patterns, this article proposes a non-intrusive machine learning method dubbed Non-intrusive Performance Anti-pattern Detector (NiPAD).***
Keywords: software performance, anti-patterns, classification, machine learning, dynamic software analysis

I. INTRODUCTION

A software performance anti-pattern is a design pattern that has a detrimental effect on the performance of a programme. It is a frequent design pattern. A large number of anti-patterns that occur often in software performance have been discovered by Smith and colleagues [1]. For example, a one-lane bridge arises when a piece of software is designed in such a way that only one or a limited number of processes may run at the same time on the same computer. As an alternative, when a software programme allocates and dealslocates memory on an infrequent basis, this is referred to as Excessive Dynamic Allocations.

When developing high-quality software systems, it is essential to identify and solve software performance anti-patterns as they emerge. Due to the fact that if an anti-pattern that is degrading application performance is not removed, system engineers will look for other solutions, such as adding more hardware to compensate for sluggish reaction times or restarting the system to address memory issues. Although there has been a long history of literature on software performance anti-patterns, source code analysis [2] continues to be the most often used technique for discovering and addressing them. Unfortunately, source code analysis requires a high degree of expertise in the subject matter. In a similar vein, it is possible that the source code will not be accessible for such scrutiny. Finally, anti-patterns in software performance may manifest themselves throughout the course of a program's execution. Source code alone makes it difficult to detect, assess, and address software performance antipatterns that may occur because of the way it is written and maintained. Another method for identifying software performance anti-patterns is to do dynamic software analysis, which is the act of analysing a software system while it is in use. Dynamic software analysis is a technique for discovering software performance anti-patterns. Current dynamic software analysis methods, on the other hand, are either architecture/deployment dependent [3] or requirement specification dependent, despite the fact that dynamic software analysis has the potential to overcome the constraints of source code analysis [4] System performance problems are assessed with the use of other dynamic software analysis techniques [5], which make use of the software instrumentation approach. In spite of the fact that these methods are useful, they do not concentrate on detecting and categorising software performance anti-patterns, which is exactly what we need. Using system performance indicators such as CPU utilisation, we present a non-intrusive method for identifying software performance anti-patterns that is not dependent on user input. This approach is designed to solve the shortcomings of the current dynamic software analysis methods, which have been discussed above in detail. The technique is referred to as Non-intrusive Performance Anti-pattern Detector (NIPAD) (NiPAD). According to NiPAD, the software performance anti-pattern identification issue may be thought of as a binary classification problem for system performance information. If the NiPAD system is provided with two sets of performance metric data, one generated from a programme that contains a software performance anti-pattern and the other generated from a programme that does not contain the anti-pattern, the system should be capable of training a binary classification technique on a given data set. The trained model is then used to predict future measurements with unknown labels, which is done on the basis of the training data (i.e., is the metric is generated from a software application that has a software performance anti-pattern). Finally, early findings from applying NiPAD to the Apache web server indicate that NiPAD can correctly identify the One Lane Bridge with a 0.94 accuracy when using Support Vector Machines (SVM) [9], with a 0.94 accuracy when using Support Vector Machines (SVM).

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 9 Issue XII Dec 2021- Available at www.ijraset.com

II. SOFTWARE EVALUATION

Software evaluation is all about running software products under known conditions with defined inputs and recording outcomes that may be compared to their set expectations.

A web-based system may be assessed in one of two ways, depending on the situation: verification or validation.

Verification ensures that the software was developed correctly and does not include any technical errors. Additionally, the verification procedure includes a review of the criteria to verify that the right problem is being handled. When software is verified, it implies that it is both syntactically and logically correct, as well as operationally correct.

On the other hand, validation entails the more difficult task of verifying that the rules' meaning and content meet a set of carefully defined criteria for appropriateness. The establishment of such criteria is essential for the validation process to be effective and for demonstrating the system's degree of acceptability to be shown. A fully formal validation is a process that begins with the establishment of validation requirements and specifications in the form of precondition requirements and design specifications, followed by the actual validation.

Additionally, problems of verification and validation are important to Databases, since they involve the creation and maintenance of a Database's correctness, by ensuring that the Database's artefacts collectively offer a realistic representation of the discourse world and its behaviour.

III. LITERATURE REVIEW

Parsons et al. [3] identified performance anti-patterns in Enterprise Java Bean (EJB) applications via the use of association rule mining. Through the use of deployment descriptors, instrumentation data, and business rules, they automate the detection of software anti-patterns. (1) Our technique is more general than prior methods because we relax the need for deployment information to be available; and (2) our approach is neither platform- or architecture-specific, which removes previously held assumptions regarding EJB semantics.

Bodik and colleagues [4] classified data centre performance crises into four categories using Logistic Regression. Similarly, Cohen et al. [11] propose a similar approach that solves the issue using Bayesian Belief Networks rather than Logistic Regression. Our work differs from theirs in that we are mainly concerned with identifying software design issues via the use of software performance anti-patterns, while their work focuses on detecting perform requirement violations.

Bodik et al. [12] utilised statistical methods to mimic Internet service workload spikes, such as a surge in search activity after the death of a renowned artist, in order to get a better understanding of how they occur. They tried to replicate this kind of job using Beta and Normal distributions, but were unsuccessful. At the cost of the proposed strategy, they are not using any categorization techniques in their research, and the emphasis of their study is on unusual occurrences of programme executions rather than on performance measures collected during routine software executions.

Khomh and colleagues [13] create BBNs from rules in order to identify code smells linked with anti-patterns in software performance. The BBN and the original source code are then used to detect and eliminate code and design smells. On the other hand, their approach is more akin to static software analysis. While our work is similar to others, it is distinguished by the fact that it is classed as a dynamic analytic technique. As a consequence, finding and categorising software performance anti-patterns using their approach is challenging.

IV. CRITERIA FOR SOFTWARE EVALUATION

A. Criteria For Software Verification

Any new system development project requires the creation of firm foundations, in this case the System's requirements. These requirements must be specified precisely in order to avoid faults in the system being developed from the bottom up. When we speak to requirements, we mean the circumstances or capabilities that a user need in order to resolve a problem or achieve a goal. System Criteria are both a description of the environment in which the system will operate and a collection of criteria that the system must satisfy in order to perform correctly. The verification criteria are intended to establish that the System's Needs include as many needs as is practically feasible given the system's scope. They are not meant to be all-inclusive.

It is the most well-known kind of verification when it comes to database integrity. In natural language, integrity is described as the characteristic of honesty, uprightness, or the original ideal state (Collins Dictionary). It has a lower meaning in the DB world, namely, plausibility, which is less essential.

Constraints on integrity are rules or conditions, or Boolean-valued functions, that must be true for a schema to be valid. As a consequence, a schema's valid instances are restricted. They are defined in such a manner that an instance is self-consistent, complete, and plausible in comparison to other examples in terms of known generic features of the discourse universe.

Databases are also concerned with the deliberate correctness of their data, which is addressed through database design and schema evolution management techniques. Rather than focusing on the data itself, the emphasis has been on avoiding data errors and inconsistencies. As a result, the theory of relational databases includes a number of semantic features such as domains and keys, column dependencies, table attributes and relationships between them, and so on, that are determined by the meaning of the data represented rather than by the database designer's design choices.

Normalisation allows the testing of database designs via the use of design constraints (a.k.a. normal forms). Typically, normal forms are defined in terms of table keys and the relationships between functional, multivalued, joinable, and template-based columns. As a consequence of these limitations, redundant data is eliminated and update anomalies are avoided. Update anomalies occur when an update creates erroneous data or accidentally deletes data.

In the case of software, verification should be conducted in terms of consistency, robustness, and correctness to ascertain the system's accuracy. When several executions of a system using the same data provide the same results or conclusions, the system is said to be consistent.

To determine which input factors are least and most significant in terms of intermediate and final results and output, robustness is assessed by submitting rules and control strategies (inference) to harsh conditions and comparing the resulting results and output. At least one test should be generated and run for each of the given system constraints. More precisely, these tests are designed to determine how the system reacts to data that is either maximum or minimum in the sense of exceeding a predefined limit. It is suggested that throughout the system testing process, the system be tested beyond the constraints imposed on it. The objective here is to discover any situations in which the system has been designed with insufficient safety margins.

A prediction's accuracy is measured by comparing the number of correct guesses to the quantity of known data. This is done to compare rule-based conclusions to historical facts and to evaluate whether or not the conclusions reached were correct, as well as how effectively the conclusions reflected reality.

V. OVERVIEW OF NIPAD

If we have two performance metric data sets from a software programme, one of which does not include a software performance anti-pattern and the other of which does, we should be able to develop a discriminant function that separates them.

These two data sets might originate from two distinct software applications or they could originate from the same software application but with two distinct configurations. On the basis of this understanding, one may divide NiPAD's approach into two phases. The classifier is first trained using data sets with a known class label (see Table 1). (i.e., has or does not have software performance antipattern). When a new performance measure is presented, the classifier is used to predict the class to which the measure will be allocated. This process is then repeated for each of the discovered software performance anti-patterns.

Table 1: Different classifiers implemented in NiPAD

Classifier	Description
Logistic Regression	Calculating the probability of a class labels.
Fisher's Linear Discriminant	Maximizing mean difference between classes and minimizing within class variance
Naive Bayes Classifier	Maximizing the class conditional probabilities using Bayes theorem
Support Vector Machines (Linear)	Maximum margin classifier methods
Support Vector Machines (RBF)	Maximum margin classifier methods, and using a kernel function

VI. PERFORMANCE METRICS CLASSIFICATION INPUTS

NiPAD uses system-level metrics as a source of information for its classifiers, which show how the application affects the system on which it is operating. System-level metrics include CPU time, memory use, and network utilisation. Because system-level metrics need less instrumentation than application-level metrics, NiPAD prefers to gather system-level data. However, before NiPAD can definitively detect software performance anti-patterns based on the network measurements, we must first determine if the data contains a enough amount of variation. This is important because it enables more precise separation of two data sets when a discriminant function is employed to do so.

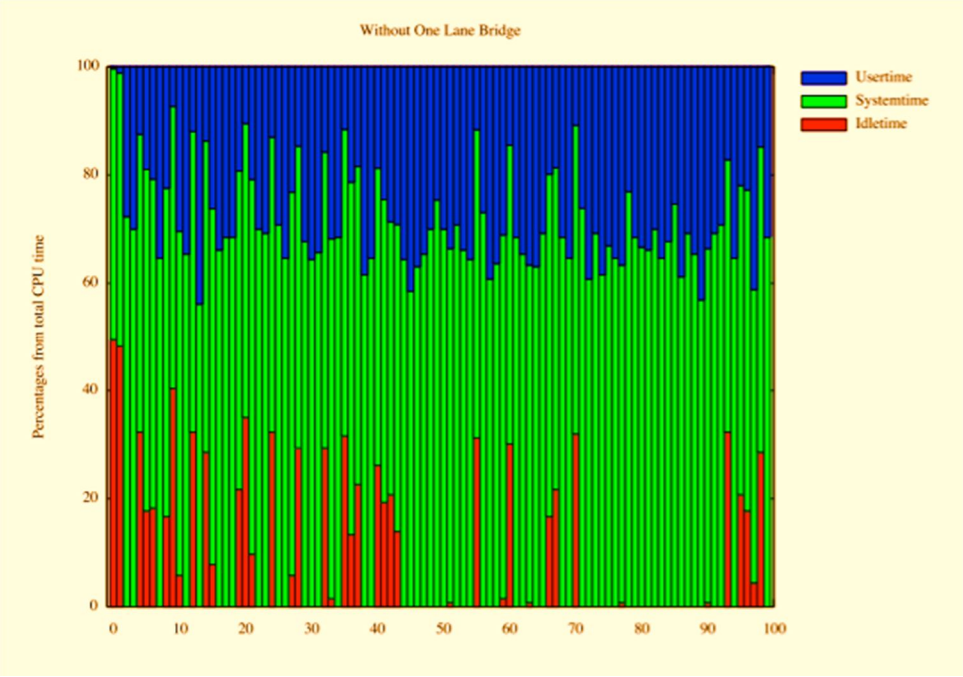


Figure 1: CPU timings without the anti-pattern of One Lane Bridge software performance

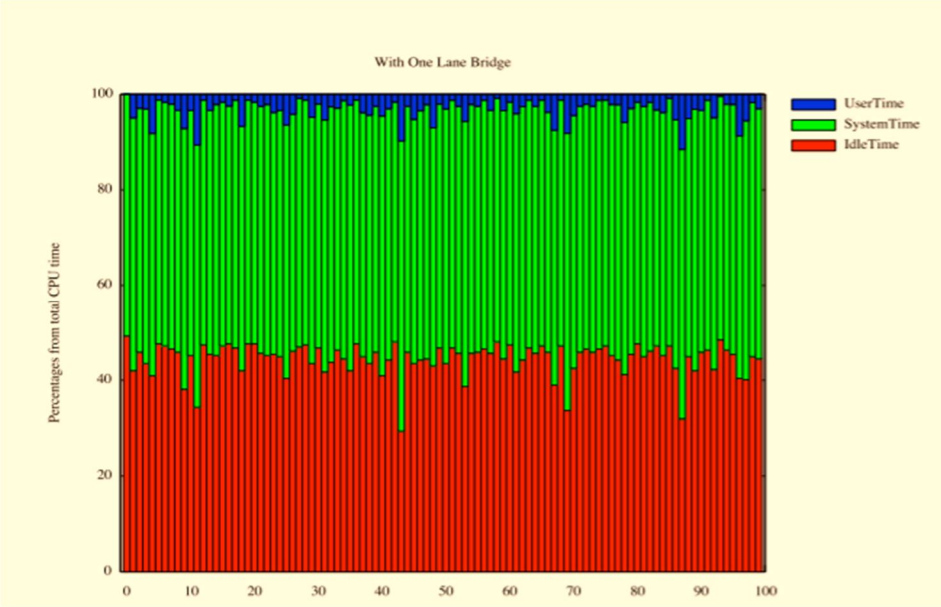


Figure 2: Without the anti-pattern of One Lane Bridge software performance, CPU timings

VII. ONE LANE BRIDGE DATA

As shown in Figures 1 and 2, the proportion of CPU time (i.e., user time, system time, and idle time) is different at various time epochs (i.e., when CPU time is sampled) when the One Lane Bridge is not present compared to when it is present in the same software programme. As shown in Figure 1, CPU idle time is low when the operating system's One Lane Bridge is disabled for the software application. On the other hand, when the One Lane Bridge is activated in the software, CPU idle time substantially rises, if not fully dominates. When a One Lane Bridge is present, the CPU time fluctuates because the software application does not make advantage of the system's parallel capabilities (e.g., all available CPU cores), causing the CPU time to fluctuate. With this fundamental insight, we are certain that NiPAD will accurately detect the (non-) existence of software performance anti-patterns such as the One Lane Bridge using system-level data.

VIII. PROBLEMS IN SOFTWARE VERIFICATION

Normalization is the process of verifying a database's correctness. This is the last step of database creation, and it was done as a necessary component of the database design process when the database was established. It is impossible to evaluate a database system's resilience in any manner while assessing it. Due to the fact that the system has already defined all of the parameters from which to choose, there are no extreme scenarios to contend with. Before using the 'Life cycle Document Management System,' all of the variables, such as the Actor, the Contract Type, and the Document Type, should be selected from a list of preset aspects. Another aspect of accuracy that this Database System cannot assess is timeliness. The system's results are very subjective, and can be verified only by comparing them to the system's previous requirements.

A. Criteria For Software Validation

Validation criteria may be qualitative or quantitative in nature. Rather of relying on subjective performance evaluations, qualitative criteria use statistical methods to compare system performance to that of the real world or a human expert. Qualitative validation criteria do not mean that the procedure is not rigorous. It is possible to develop a highly formalised qualitative validation procedure. Several instances of qualitative validation criteria are as follows:

Generality: Generality is measured in terms of a system's ability to be applied to a wide number of similar problems. The generality of a system refers to the range of possible operating conditions. To be called broad, a system must be capable of dependably functioning in a wide range of situations and contexts.

Adaptability is measured in terms of a system's ability to be customised to suit particular user needs, as well as its future development and implementation possibilities. In addition, the system must be adaptable to a range of work situations and link with existing or future hardware and software systems.

Compatibility tests are used to evaluate if the new system is capable of absorbing the capabilities and modes of operation of the legacy system under the circumstances envisaged. In other words, a search for incompatibilities across operating systems is performed.

Interaction with Visual Environments: (or user interface) Visual interaction may be evaluated by assessing the system's user friendliness and seeing how a human user interacts with it. A quantitative evaluation of the interface's readability and usability must be conducted in accordance with the quantifiable metrics specified in the System Specification. During security testing, attempts are made to circumvent the system's security, such as access to a database owned by an unauthorised user.

In certain cases, a system will be designed with particular maintenance requirements, such as the condition that changes of a given degree of complexity take no longer than a specified period of time.

We are particularly interested in systems that can be installed on a variety of different machines and models, as well as systems that enable a variety of different installation options, such as the use of various peripheral devices. Before a system with specified storage requirements, such as a maximum amount of main or backing storage occupancy, can be considered operational, we must develop tests that look for instances where the system exceeds the specified limits, such as when processing or supplying large amounts of data in a manner similar to that used in the volume tests.

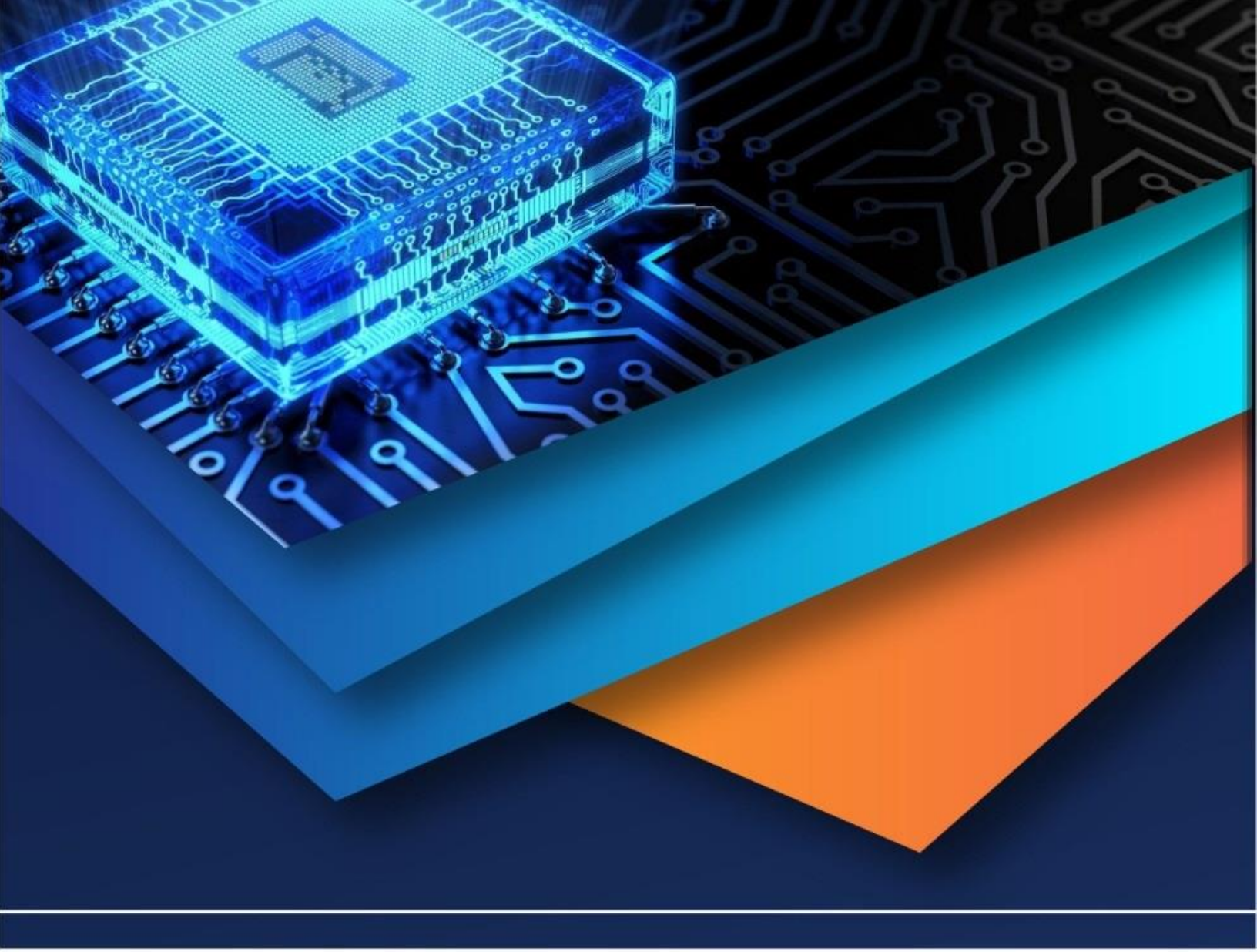
Usability: The prototype system's usability, as well as interoperability with existing web applications, is important in deciding its acceptance and usage. When developing a prototype system, it is essential to keep the user in mind. This is particularly true during the early phases of development, while developing a prototype system that makes use of basic facilities, does not need complex construction, and does not require users to expend considerable effort transporting or utilising the prototype system.

IX. CONCLUSION

The system was verified to verify that it fulfilled all requirements and was free of technical flaws. It was determined that various academics with expertise in architecture would be charged with the responsibility of reviewing and validating the database's contents and ideas. As a consequence of the different views presented, the system was improved and reorganised. Additionally, a validation procedure was conducted to solicit feedback from potential system users. With tremendous success, all organisations who had the necessary infrastructure in place to support WPMS felt that the system was very beneficial, not just for their contacts with other businesses that used WPMS, but also for their internal document management. Students' opinions and suggestions were considered and included into the curriculum. As a consequence of this study, we have framed the topic of software performance anti-pattern detection as a binary classification problem and forecasted unlabelled performance data using machine learning classification techniques. According to our experiments and results, software programmes are often used in conjunction with other applications, which introduces noise into the data stream.

REFERENCES

[1] U. Smith and L. G. Williams, "Software performance antipatterns," in Proceedings of the 2nd international workshop on Software and performance. ACM, 2000, pp. 127–136.
[2] V. Cortellessa, A. Di Marco, and C. Trubiani, "An approach for modeling and detecting software performance antipatterns based on first-order logics," Software & Systems Modeling, pp. 1–42, 2012.
[3] T. Parsons, "A framework for detecting performance design and deployment antipatterns in component-based enterprise systems," in Proceedings of the 2nd international doctoral symposium on Middleware. ACM, 2005, pp. 1–5.
[4] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacentre: automated classification of performance crises," in Proceedings of the 5th European conference on Computer systems. ACM, 2010, pp. 111–124.
[5] R. P. Erkki Salonen, "Find the bug, Fix the bug, do it fewer times (TimeToPic)," http://www.timetopic.net/Pages/default.aspx, 2012.
[6] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in Symposium on Networked Systems Design and Implementation. USENIX Association, 2012, pp. 26–26.
[7] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in Proceedings of the 2012 International Conference on Software Engineering. IEEE Press, 2012, pp. 145–155.
[8] J. H. Hill, H. A. Turner, J. R. Edmondson, and D. C. Schmidt, "Unit Testing Non-functional Concerns of Component-based Distributed Systems," in Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation, Denver, Colorado, apr 2009, pp. 406–415.
[9] Cortes and V. Vapnik, "Support-vector networks," Machine learning, vol. 20, no. 3, pp. 273–297, 1995.
[10] D. Berger, B. G. Zorn, and K. S. McKinley, Reconsidering custom memory allocation. ACM, 2002, vol. 37, no. 11.
[11] Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," in Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, vol. 6, 2004, pp. 16–16.
[12] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in Proceedings of the 1st ACM symposium on Cloud computing. ACM, 2010, pp. 241–252.
[13] Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in Quality Software, 2009. QSIC'09. 9th International Conference on. IEEE, 2009, pp. 305–314.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY