SmartHeal: Real Time Automation of Software Errors

Mr. D. Madhu Sudan Reddy
Computer Science and Business System
B V Raju Institute of Technology
NARSAPUR, India
madhusudhan@bvrit.ac.in

Hemanjali Talasila
Computer Science and Business System
B V Raju Institute of Technology
NARSAPUR, India
22211a3254@bvrit.ac.in

Akhila Joshi
Computer Science and Business System
B V Raju Institute of Technology
NARSAPUR, India
22211a3202@bvrit.ac.in

Yugender Ballari
Computer Science and Business System
B V Raju Institute of Technology
NARSAPUR, India
22211a3261@bvrit.ac.in

***Abstract*—The self-healing software system is designed to automatically find and fix the anomalies. Using sophisticated monitoring and anomaly detection algorithms, the system can identify deviations from the usual behaviour in real-time and automatically implement corrective measures. Machine learning models augment this by detecting impending failures and addressing them beforehand, reducing downtime and enhancing system reliability. The system functions by continuously monitoring the system metrics and application activity. When detected anomalies, it triggers remedial actions such as terminating the process, or giving suggestions to user's intervention. The process guarantees smooth functions with minimal disruption, allowing the system to function optimally without depending much on manual supervision. This paper suggests that the system learns from the past behaviour to improve its anomaly detection and healing procedures so that it can handle multitude of problems efficiently. As the system matures, it becomes more effective at identifying likely failures and reacting faster effectively, further improving its general dependability.**

Keywords— Self-healing, Anomaly Detection, Machine Learning, Automation

I. INTRODUCTION

In contemporary computing setups, especially those including cloud computing, edge computing, and distributed systems, reliability and resilience are crucial for better performance and productivity. Conventional methodologies for failure management like manual intervention or scheduled maintenance are not adequate to guarantee uninterrupted operation in these dynamic conditions [1]. Thus, self-healing Software systems have become an intelligent response to autonomically detect, diagnose, and mitigate system failures in real-time and minimizing the need for human intervention and improving system availability [2].

The core of the self-healing system is the capability to constantly monitor system health by gathering real-time operational data. It comprises key system metrics like CPU usage, memory usage, and individual process behaviour [3]. Using this information, self-healing systems can identify anomalies which appear to be different from the normal operational patterns that can cause failures, performance degradation, or resource bottlenecks. Anomalies are identified via a machine learning model like the Isolation Forest algorithm, which can the outlier classification system and the prediction of possible failures before they occur.

Furthermore, self-healing systems also use past data to analyse and find historically heavy applications which are likely to use up excessive resources over longer periods. Analysing past resource usage patterns, can help in predicting when an application is likely to go over acceptable thresholds and incorporate proactive measures to ensure there is no system degradation [4]. For instance, applications that previously exhibited a tendency to generate excessive CPU or memory consumption can be marked as historically heavy, and the system can utilize remediation techniques to deal with such applications efficiently.

When anomalies are encountered whether in real-time monitoring or historical analysis, the self-healing system implements corrective measures based on pre-established thresholds and severity levels [5]. The actions can vary from shutting down resource-intensive applications to more refined techniques like process prioritization, cache clearing, or restarting services. The system prompts the user with the options to shut down the troublesome application or selecting an alternative measure like process prioritization, clearing of cache, or restarting a specific process or service, thereby maintaining the user's control while ensuring system stability [6].

Also, self-healing systems include cooldown mechanisms to avoid overreacting to transient problems. By introducing time-based delays, the system avoids terminating applications prematurely, which would interfere with the user experience. The cooldown mechanism ensures that the appropriate corrections are only made when absolutely necessary, encouraging system stability and customer satisfaction.

By combining such capabilities, self-healing systems reduce downtime and also adapt dynamically to the changing conditions of contemporary computing environments. This paper examines the architecture, methodologies, and implementation strategies behind such self-healing mechanisms, underlining the application of machine learning for anomaly detection and a historical trend analysis for proactive resource management [7]. We examine the efficacy of cooldown mechanisms for avoiding unnecessary interruptions and maintaining a balance between responsiveness and stability. Based on rigorous analysis and practical research, we show how effective failure mitigation can substantially improve the dependability, scalability, resilience of cloud-based, distributed, and edge computing environments.

II. LITERATURE REVIEW

Chandola, Banerjee, and Kumar (2009) provide a comprehensive survey of anomaly detection techniques which provide the basis for self-healing software systems [7]. They classify anomaly detection techniques into statistical, machine learning-based, and proximity-based methods, highlighting their applications in cybersecurity, fraud detection and system monitoring. The article emphasizes the limitations of conventional rule-based anomaly detection methods and highlights the increasing relevance of AI driven solutions, which are, neural networks and ensemble learning, to enhance detection accuracy. Their work is particularly relevant to self-healing software systems as it provides insights into various anomaly detection techniques that can be leveraged for real-time fault detection [7].

Ghosh, Weiss, and Ramachandran (2018) explore the evolution of self-healing software systems and propose a framework that integrates monitoring, diagnosis, and automated recovery. They discuss the challenges associated with implementing self-healing capabilities, including scalability, adaptability, and security concerns. Their study presents case studies of existing self-healing systems, illustrating how AI and machine learning can be employed to predict failures and trigger automated corrective actions. The paper's key contribution lies in its emphasis on predictive self-healing, where ML models learn from historical data to anticipate and mitigate failures proactively, making it highly relevant for modern autonomous systems [8].

Salehie and Tahvildari (2009) examine self-adaptive software systems, which share common principles with self-healing systems. They classify adaptation techniques into rule-based, learning-based, and architecture-based approaches and discuss the importance of feedback loops in enabling dynamic adjustments. Their research throws light on the trade-offs between flexibility and reliability in self-adaptive software and raises a number of research problems, including uncertainty management and decision making complexity. Offers a theoretical underpinning for self-healing software by outlining various adaptation strategies and their relevance in different computing environments [9].

Wang, Zhang, and Li (2020) perform a comprehensive review of reinforcement learning (RL) methods for self-healing software systems. They examine how RL-based agents can learn autonomously optimal actions for recovery by interacting with the environment, minimizing the need for human intervention. The paper covers different RL algorithms, including Q-learning and deep Q-network (DQN), assesses their capability in reducing system failures and resource allocation optimization. Their conclusions emphasize the capability of learning from their past experiences and context-sensitive recuperation mechanisms, which make it an effective method for AI-based fault management [10].

Weyns, Malek, and Andersson (2012) analyse decentralized self-adaptation strategies, which are particularly vital for large-scale distributed systems. They contend that centralized self-healing mechanisms tend to encounter scalability and performance bottlenecks, while decentralized strategies enhance fault tolerance and resilience. Their paper shows real-world case studies illustrating how decentralized control loops can be used to increase cloud self-healing computing and edge computing infrastructure. This paper is useful for developing scalable self-healing designs that can function well in complicated as well as changing systems [11].

III. METHODOLOGY

The methodology is centered around developing a self-healing system by integrating real-time system monitoring, labelling in accordance with established performance standards, machine learning-based anomaly detection. The process begins with ongoing collection of system-wide and process-specific resource usage measures, which are systematically logged and stored for further processing. The gathered information is then classified based on CPU and memory usage patterns, with mainline usage as well as resource-intensive anomalies.

Machine learning model based on the Isolation Forest algorithm is trained on this labelled data to recognize deviations from typical performance [12]. Lastly, the trained model is coupled with a monitoring script in real-time to raise alarm and mobilize an appropriate response and healing actions guided by users [2][8]. The structured methodology ensures the system develops with time, which enables ongoing refinement through iterative model retraining and data collection.

A. Data Collection and Monitoring

Data collection process within the system is carried out in two primary aspects: system-wide metrics monitoring and process specific monitoring.

a) System Metrics Collection: The system metrics are monitored by a script written in Python which uses the psutil library. It measures the CPU usage and memory usage at a one-second interval. Each sampled data point is linked with a UNIX timestamp and is recorded for time series analysis. With this, the system is enabled to temporary peaks or anomalies in system behaviour.

b) Process Monitoring: In addition to system metrics, detailed information about each running process is gathered. The script loops over all running processes and filters out only the process that are spending over 1% of the CPU. For every process chosen, the following attributes are included: Process ID(PID), Process Name, CPU Usage, Memory Usage, Process Priority are obtained.Priority (or nice value) is included to comprehend the preference for execution by process by the operating system's scheduler.

c) Data Logging: The gathered information is organized in rows and saved as a CSV file. If such a file does not exist, it is formatted with the same style as existing entries. This logging operation occurs at a regular five second interval, creating a continuous and high-resolution dataset for subsequent analysis.

B. Data Preprocessing and Labeling

The following phase is the pre-processing of the gathered dataset and marking entries as per the system's functionality and operational health.

a) Loading the Collected Data: The data is loaded from the metrics_new.csv file into a pandas DataFrame. This structured dataset contains columns for timestamp, CPU usage, memory usage, and process-specific details including CPU, memory, and priority values.

b) Anomaly Labeling: Each data entry is classified based on predefined resource utilization thresholds. If the system-wide CPU usage exceeds 90%, the status is labeled as High CPU Load. If memory usage exceeds 85%, the status is set to High Memory Usage. If any individual process exceeds 50% CPU usage, the status is designated as High Process Usage. All other entries are marked as Normal, reflecting stable operation conditions [4].

c) Detection of Historically Heavy Applications: To capture persistent resource hogs, the system performs historical analysis. Applications that consistently exhibit high CPU (average process CPU usage above 40% or memory usage (average process memory usage above 20% are classified as historically heavy applications. Processes satisfying these criteria are marked with an "is_heavy_app" flag and assigned the Historically Heavy App status.

d) Defining Healing Actions: Based on the assigned status, specific healing actions are mapped. Instances labeled High CPU Load are associated with restarting critical processes. Entries under High Memory Usage suggest freeing up memory or closing unused applications. For High Process Usage, the recommendation is to terminate the resource-heavy process. In cases of Historically Heavy App, the system suggests either terminating the heavy application or offering the user an alternative. Normal entries require no action. All labeled and action-mapped data is saved into a new CSV file titled labeled_metrics_new.csv.

C. Machine Learning Model Training

Following labeling, the system transitions to training an anomaly detection model to automate the identification of abnormal states.

a) Feature Selection: From the labeled dataset, four critical features are selected for model training: system CPU usage, system memory usage, process CPU usage, and process memory usage. These numerical features are extracted into arrays suitable for feeding into a machine learning model.

b) Model Selection and Training: The Isolation Forest algorithm is chosen due to its effectiveness in high-dimensional anomaly detection tasks. A contamination rate of 0.01 is configured to account for the rarity of true anomalies within the dataset. The model is trained exclusively on Normal labeled entries to ensure that it learns the baseline healthy behavior of the system.

c) Model Storage: Once trained, the Isolation Forest model is serialized and saved as a .pkl file using the joblib library. This allows the model to be efficiently loaded later during the real-time monitoring and anomaly detection phase.

D. Real-Time Anomaly Detection and Self-Healing

This phase integrates the trained anomaly detection model into a real-time monitoring pipeline capable of initiating intelligent, user-guided healing actions. The system architecture ensures continuous resource observation, anomaly classification, user interaction, and controlled response timing.

a) Model Loading and Initialization

At runtime initialization, the system loads a pre-trained Isolation Forest model serialized in a .pkl format. This model, trained on structured CPU and memory usage metrics, serves as the core inference engine. The model loading process is encapsulated within exception-handling routines and is supported by comprehensive logging to track load status and ensure system observability.

b) Continuous Real-Time Monitoring

System-wide and application-level metrics including CPU usage, memory consumption, process identifiers (PIDs), and executable names—are collected in real time using platform-independent monitoring libraries. These metrics are preprocessed into a normalized vector format consistent with the training data schema to enable accurate anomaly inference.

c) Anomaly Detection

The monitoring engine periodically feeds the preprocessed metrics into the Isolation Forest model to detect deviations from learned normal behavior [7]. A prediction label of "-1" indicates an anomalous process instance. The system flags these anomalies for further evaluation and possible corrective action.

d) Interactive Healing with Cooldown Management

Upon detecting an anomaly, the system presents a non-blocking graphical prompt to the user via a Tkinter-based interface. This interface offers a binary decision: proceed with healing or ignore the alert. If the user consents, a secondary interface presents multiple healing strategies:

- Terminate Process: Immediately ends the application using system-level termination commands.
- Restart Process: Attempts to close and relaunch the application executable.
- Reduce Priority: Dynamically lowers the CPU priority of the process.
- Pause and Resume: Temporarily suspends the process and allows the user to resume it later.
- Cancel Action: Allows the user to decline without executing any action.

To prevent user fatigue and repetitive prompts, the system maintains a cooldown registry. After a prompt is triggered for a particular process, that process is suppressed from further prompting for a fixed cooldown interval for 5 minutes, even though monitoring continues for all other applications. This mechanism ensures responsive yet non-intrusive operation across diverse system workloads.

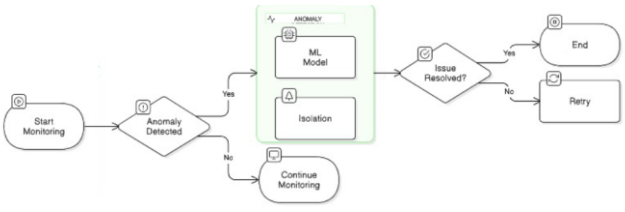

Fig. 1. Anomaly Detection and Self-Healing Decision Process

E. Continuous Monitoring and Iterative Learning

The system is designed and developed to run in an ongoing loop of monitoring, labelling, and learning.

a) Continuous Data Collection: With the system operating, new performance measures continue to be

gathered and added to the dataset so that the model can learn to accommodate changing workload patterns and application behavior.

b) Retraining of the future model: As the accumulation of fresh labelled data permits periodic retraining of the Isolation Forest model. Retraining can improve detection accuracy with consideration of changes in system behavior, newly installed programs, or upgrades to the operating system.

c) Reinforcement Learning Integration: The Future enhancements are also envisioned to include reinforcement learning strategies. Such strategies would allow the system to not only identify and recommend remedying measures, but autonomously perform the best healing actions-based reward mechanisms not involving explicit users.

IV. RESULTS AND DISCUSSIONS

The self-healing software system watches over and scans real-time system measurements, such as CPU usage, memory consumption, and active processes. It records these metrics at 5 second intervals capturing variations in resource usage over time. It records processes over the course of consuming $\geqslant$ 1% CPU, ensuring that only substantial contributions to system loading are taken into account. The data gathered is tagged with particular anomaly categories such as High CPU Load, High Memory Use, and High Process Use enabling a well-focused and correct examination of possible system failures. This labeled dataset is utilized for model training, wherein an Isolation Forest algorithm is used with four chosen characteristics. The learning process is finished successfully, and the model is saved for subsequent deployment in anomaly detection tasks.

After integrating the model, the system continuously conducts real-time monitoring of CPU and memory usage while monitoring all running processes. When the software detects an anomaly in a specific application, which immediately activates a request for user confirmation, with a pop-up alert to notify the user of the problem. If the user acknowledges the anomaly, the system initiates corrective action by stopping the affected application and putting it on a cooldown list. This cooldown mechanism prevents the system from continuously flagging and closing the same application in a short time period eliminating unnecessary interruptions and optimizing decision-making. These actions are captured in monitoring logs with timeframes, which gives a good idea of when anomalies arise, how they are dealt with, and if so, Corrective actions are implemented.

TABLE I. SYSTEM MONITORING LOSS

Timestamp	CPU Usage (%)	Memory Usage (%)	Action User Processes	Status
22:51:18	19.7%	60.2%	None	Normal
22:51:27	18.1%	60.2%	Scanning	Normal
22:51:27	19.1%	2.25%	Anomaly Detected	Alert Triggered
22:51:43	-	-	Application added to Cooldown	Cooldown Initiated

The Isolation Forest model exhibits robust and stable performance with 87.49% model accuracy, which shows Effective identification of normal and abnormal system behavior. The 0.94 ROC-AUC score indicates a high ability to differentiate between normal cases and anomalies. Also, the 87.08% accuracy of cross-validation verifies uniform performance in various subsets of the dataset, validating its appropriateness for real-time anomaly detection.

The performance of the system was measured using of a confusion matrix, which gives insights to its ability to differentiate normal versus anomalous behaviour. The results demonstrate that the model accurately identified 30,793 anomalies with a total of 274 false negatives, illustrating High accuracy of detection. The model, however, also produced 4,384 false positives, in which normal applications were mistakenly labelled as anomalies. This indicates that although the model is effective in screening for outliers, further Threshold optimization and selection of the right features may enhance its accuracy.

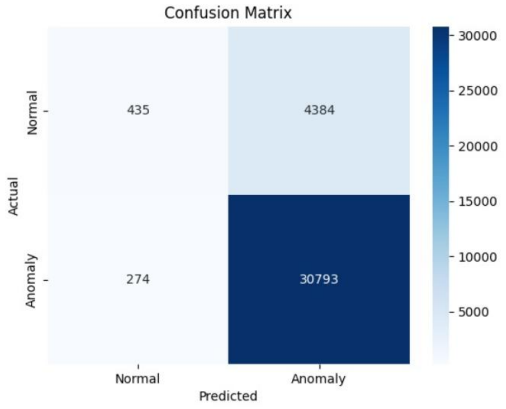

Fig. 2. Confusion Matrix

In addition to line charts for CPU and Memory usage trends, logged process bar charts fluctuations, and a heatmap-based confusion matrix designed to offer a graphical overview of system performance. These visualizations demonstrate how resource changes in consumption over time, how often new Processes are tracked, and the accuracy with which the model predicts system behavior. In general, the execution findings affirm that the self-healing mechanism is able to sense and act to system anomalies in an automatic way, improving system stability as well as minimizing possible interruptions. The improvements include fine-tuning model parameters involving adaptive learning strategies and incorporating a Feedback loop that continuously refines the system's anomaly detection ability.

TABLE II. CONFUSION MATRIX

Actual/Predicted	Normal	Anomaly	Total
Normal (0)	435	4384	4819
Anomaly (1)	274	30793	31067
Total	709	35177	35886

V. CONCLUSION

The self-healing software system effectively monitors and deals with real-time system inconsistencies, considerably enhancing system stability and reducing the necessity for manual intervention. By constantly monitoring critical metrics like CPU and memory usage, as well as Monitoring active processes, the system can immediately detect and categorizing anomalous behavior with the Isolation Forest model.

The significance of self-healing mechanisms is particularly visible in situations where system availability is paramount. In cloud infrastructures, distributed networks, and containerized applications in the event of downtime, loss of data, and security exposures, all of which can have important financial and operational impacts. autonomous control of faults and starting corrective actions, self-repairing software systems enable maintaining high availability and system performance ensuring that applications continue to function even under conditions of peak load or unexpected failures.

REFERENCES

[1] G. D. Rodosek, K. Geihs, H. Schmeck, and B. Stiller, "Self-healing systems: Foundations and challenges," in Self-Healing and Self-Adaptive Systems, 2009.

[2] F. M. David and R. H. Campbell, "Building a self-healing operating system," in Proc. Third IEEE Intl. Symp. on Dependable, Autonomic and Secure Computing (DASC 2007), IEEE, 2007, pp. 3–10.

[3] P. Koopman, "Elements of the self-healing system problem space," 2003.

[4] H. Psaier and S. Dustdar, "A survey on self-healing systems: Approaches and systems," Computing, vol. 91, pp. 43–73, 2010.

[5] H. Naccache and G. C. Gannod, "A self-healing framework for web services," in Proc. IEEE Intl. Conf. on Web Services (ICWS 2007), IEEE, 2007, pp. 398–345.

[6] G. Salvaneschi, C. Ghezzi, and M. Pradella, "Contexterlang: A language for distributed context-aware self-adaptive applications," science of Computer Programming, vol. 102, pp. 20–43, 2015.

[7] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly survey," ACM Computing Surveys, vol. 41, no. 3, pp. 1–58, 2009.

[8] S. Ghosh, G. Weiss, and U. Ramachandran, "Self-healing software systems: Survey and directions for the future," Journal of Systems Software, vol. 140, pp. 111–135, 2018.

[9] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and Research Challenges," ACM Transactions on Autonomous and Adaptive Systems, vol 4., no. 2, pp. 1–42, 2009.

[10] X. Wang, Y. Zhang, and Y. Li, "Reinforcement learning for self-healing software systems: A review," IEEE Access, vol. 8, pp. 21056–21072, 2020.

[11] D. Weyns, S. Malek, and J. Andersson, "On decentralized self-adaptation: Lessons from the trenches and challenges for the future," ACM Transactions on Autonomous and Adaptive Systems, vol. 7, no. 4, pp. 1–32, 2012.

[12] W. Xu, X. Zhang, and J. Wang, "System anomaly detection using machine learning: A survey," Journal of Software Engineering, vol. 30, no. 5, pp. 567–590, 2016.