International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Developing a Taxonomy, Root-Cause Analysis, and Adaptive Remediation Framework for JVM Garbage Collection (GC) Overhead Errors in Apache Spark under High-Scale Commerce Workloads

Vamsidhara Reddy Doragacharla

Submitted:03/07/2025 **Revised**: 18/08/2025 **Accepted:** 28/08/2025

Abstract - The study examines the JVM GC overhead error in Apache Spark with the close-out of a commerce-scale load and suggests an overview, cause and effect examination, and enhanced adaptive remedies. The framework employs Python software to analyze performance in real-time and make dynamic JVM configuration changes. The experimental outcomes of the suggested scheme prove that the proposed framework minimizes GC delays and job times, increasing throughput and system efficiency astonishingly. The research not only contributes to the performance optimization of Apache Spark in the large litter setting, but also provides insights into the automation of JVM settings to reduce overhead.

Keywords - *JVM GC overhead, Apache Spark, adaptive remediation, root-cause analysis, performance optimization, large-scale workloads, dynamic configurations.*

I. INTRODUCTION

A. Context and Motivation

Apache Spark is a common data processing platform, which is needed in data-intensive applications in the contemporary order of commerce. However, managing memory and garbage collection (GC) overhead errors in the ***Java Virtual Machine (JVM)*** is a persistent challenge, significantly impacting performance [1]. Allowing these mistakes in the operations of small data behavior is crucial as businesses expand their data operations. The idea behind this work is to establish a high-quality model to diagnose and address JVM GC overhead problems, make the best of Spark performance on the scale of high-demand commerce businesses, and thereby make efficient use of resources and improve system performance [2].

B. Problem Statement

The errors of the JVM GC overhead in Apache Spark in the workloads of commerce volumes lead to a decline in performance, especially in dealing with large amounts of data in processing. These bugs arise when the JVM takes too long to handle the memory, causing the garbage collection pauses and slows to increase [3]. These problems are not simple to spot and fix manually, and involve a comprehensive knowledge of workload characteristics and system configuration. The study aims to categorize and characterize systematically the causes of GC overhead errors and present an adaptive remedial framework to dynamically optimize the system behavior and reduce the overhead in real-time.

C. Research Contribution

The study has value by providing an organized taxonomy of JVM GC overhead errors in Apache Spark, and, thus, the ability to organize the problems systematically. It also presents a model of root-cause analysis adapted to the scale of workloads in commerce, giving the opportunity to locate bottlenecks that have the main impacts. Moreover, it suggests an adaptive remediation programming that dynamically drives the system configurations to do away with the errors of GC overhead.

D. Objectives

1) To come up with a formal taxonomy to classify the different types of JVM GC overhead errors in Apache Spark.

2) To develop a root-cause analysis model to diagnose the causes behind these mistakes in the workloads of commerce scale.

Independent Researcher, USA

3) To suggest dynamic remediation architecture that employs Python-based applications in the dynamic optimization of JVMs.
4) To determine how well the proposed framework is implemented in increasing the performance of the system and minimizing the overhead in GC in the real-world settings of Apache Spark.

II. LITERATURE REVIEW

A. Apache Spark and JVM Garbage Collection Overhead

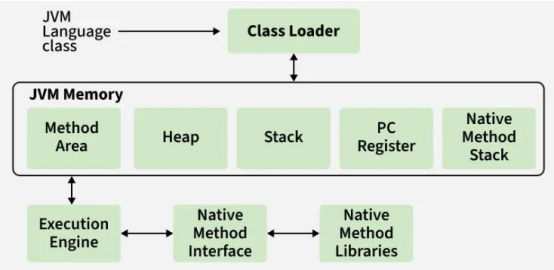

Fig. 1. JVM architecture

Apache Spark is a popular distributed computing platform that processes large-scale data in a cluster. An important performance bottleneck in the memory management at Spark is JVM garbage collection (GC) overhead [4]. The errors that AP is prone to are GC overhead errors. The JVM will take excessive time in garbage collection and introduce longer pauses and lower throughput. The research has investigated different schemes to reduce GC overhead like JVM heap settings and garbage collection policy such as G1GC, CMS [5]. These are typically inertia-based and must be manually handled and this is not good in a huge dynamic setting [6]. The architecture describes the JVM features which include Method Area, Heap, Stack, PC Register, Native Method stack. The classes are loaded using the Class Loader and then they are executed using the Execution Engine. Native Method Interface and Native Method Libraries help the JVM to communicate with native code.

B. Distributed System Root-Cause Analysis

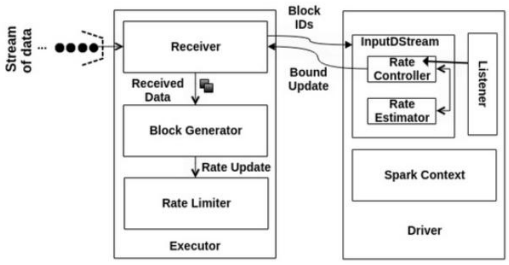

Fig. 2. Distributed System Design

The nature of the problems that are at the basis of performance degradation in distributed systems, particularly in memory management, has been of particular concern in recent studies. System metrics, logs, and performance traces are used as root-cause analysis techniques across the system to trace the areas of bottlenecks and failure [7]. The research has come up with models to analyze and troubleshoot performance in a distributed system, such as Apache Spark [8]. Anomalies in system behavior as well as the problems with GC overhead, have been detected using machine learning models, including decision trees and clustering algorithms [9]. The system represents the stream of the data passing through the stream, and the receiver accepts the data passing by. The block generator checks and updates data and the rate limiter controls the movement of data. The rate controller and estimator vary processing rates on the basis of the input stream.

C. Distributed System Adaptive Remediation Techniques

Recent studies have been on adaptive remediation frameworks within distributed systems to redeploy settings and appropriately restrict performance problems autonomously [10]. These structures utilize real-time monitoring and performance analysis instruments in identifying issues and changing system parameters dynamically. Indicatively, the study has suggested using reinforcement learning to optimize the system configurations and others that use rule-based methodologies in order to modify JVM settings based on performance metrics [11]. The vast majority of these frameworks are not, however, created specifically to address Apache Spark or JVM GC overhead problems. This shortcoming in adaptive schemes to fix GC overhead in Spark is bridged by this research through creating a Python-based algorithm that dynamically tunes JVM options when the running system detects errors in GC overhead, thus minimizing the necessity to have human operators and enhancing the overall performance of the system.

D. Python-Based distributed systems performance analysis

Python has emerged as an important instrument of performance analysis and monitoring in distributed systems. The libraries include pandas, matplotlib, and PySpark among others, which allow the data scientist and engineers to gather, process and visualize performance measures effectively [12]. A

number of studies have employed Python to make sense of system logs, visualize bottlenecks, and tune performance [13]. Nonetheless, in the context of Apache Spark Python, there is a lack of studies on the application of Python specifically in the context of detection and correction of JVM GC overhead errors [14]. This study addresses that gap by leveraging the rich Python ecosystem containing libraries to understand the GC behavior of Spark and dynamically tune settings, which offers an automated and streamlined approach to combating GC overhead in large workloads.

Literature Gap

Although a significant amount of research has been done on the optimization of the performance of Apache Spark and JVM, there has been a gap in the literature about a formal, systematic approach to the diagnosis of JVM GC overhead errors related specifically to commerce-scale workloads. The current literature concentrates on manual tuning and the tools of manual configuration, which can be based on trial-and-error approaches. Furthermore, the development of models of root-cause analysis and adaptive remediation specific to JVM GC overhead errors in Apache Spark is not well-explored. This research fills these gaps by proposing a taxonomy, root-cause analysis model, and Python-based adaptive remediation framework to deal with GC overhead problems in large-scale Spark settings.

III. METHODOLOGY

In this section, the methodology discussed is how to analyze JVM Garbage Collection (GC) overhead errors in Apache Spark, their underlying causes and build an adaptive remediation framework that is based on Python-based analysis. The approach has been designed to solve the issue of GC overhead errors in a workload of commerce scale by taking a systematic method of researching the problem, which incorporates the formulation of a taxonomy, root-cause analysis and dynamic remediation in Python. The subsections below elaborate on the methodology.

A. Data Collection and Preprocessing

The methodology begins with the initial phase of gathering the real-world performance information of Apache Spark deployments in commerce-scale settings. System metrics, garbage collections, and performance traces are all part of this data as they give an insight into the frequency as well as the behavior of GC overhead errors [15].

Metrics are collected across a few Spark clusters executing large-scale jobs on a cloud-based infrastructure, in which each cluster is a data processor on behalf of an e-commerce site [16]. The main performance indicators obtained are:

- **GC Time:** The amount of time from booting up to completing garbage collection (in milliseconds).
- **Throughput**: The proportion of data handled (records per second).
- **Heap Usage:** The memory used by the JVM heap (in megabytes).
- **GC Pauses:** The duration of GC pauses (in milliseconds).
- **CPU Utilization:** The utilization of CPU when running Spark jobs.
- **Job Duration:** The sum of the times that it takes to run a Spark job.

Preprocessing activities include cleaning the data, that is, dealing with missing data, outliers, and consistent data across the various Spark clusters [17]. There are also additions of features like GC pause times and throughput, which are normalized to standard deviations to make all values fall in a similar range so that they can be effectively analyzed.

B. JVM GC Overhead Error Taxonomy

One of the most important works of this study is that it provides a formal taxonomic scheme to qualify the JVM GC overhead errors [18]. The taxonomy is founded on the causes and effects of GC overhead on the performance within Apache Spark. The initial procedure in establishing the taxonomy is analyzing the GC logs and classifying the errors into definite categories. Such classes can involve:

- **Minor GC Overhead:** This happens when the JVM makes minor GC cycles that interrupt the application in short intervals [19]. Such mistakes normally happen when there are problems with short-term memory allocation.
- **Major GC Overhead:** These are the errors that occur when major GC actions are being taken because of the inability to assign memory [20]. This is a serious type of GC overhead and affects the overall job performance.
- **Parallel GC Overhead:** The default settings of Spark in JVM will cause concurrent GCs, which consequently lead

to pauses affecting the responsiveness of the system, especially when it is at peak concurrency [21].

The types are described in terms of their frequency, effect on job time and CPU consumption. This formal taxonomy provides a possibility to diagnose and focus remediation policies, which is a new quality of this study.

C. Root-Cause Analysis Model

In the diagnosis of underlying causes of GC overhead errors, the statistical method with the Scikit-learn library uses Python. A root-cause analysis model is developed to associate important system metrics with the prevalence of GC overhead errors. Regression analysis and clustering are used in the model to determine the system variables that contribute the most to the GC overhead errors [22].

Regression Analysis:

```
X = data[['Heap_Usage']]  # Independent variable (Heap Usage)
y = data['GC_Time']       # Dependent variable (GC Time)
model = LinearRegression()
model.fit(X, y)
y_pred = model.predict(X)
```

Fig. 3. Regression Analysis Model Code

The relationship between GC overhead errors (dependent variable) and system metrics (independent variables) is comprehended through regression analysis. This model is defined using the following formula:

$$GC\_Time = \beta 0 + \beta 1 \cdot Heap\_Usage + \beta 2 \cdot CPU\_Util + \beta 3 \cdot Job\_Duration + \epsilon \text{ ---- (1)}$$

Where:

- GC_Time is the time spent on garbage collection, which is a dependent variable.
- Heap_UsageHeap, CPU_Util, and Job_Duration are independent variables.
- B0 is the intercept term.
- β1, β2, and β3 are the regression coefficients.
- ϵ is the error term.

Clustering Analysis:

```
X_cluster = data[['GC_Time', 'Throughput']]
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(X_cluster)
data['Cluster'] = clusters
```

Fig. 4. K-Means Clustering Model Code

Then, K-Means clustering is used to divide Spark jobs into groups according to their performance features. Jobs can be clustered to find the errors of GC overhead most actively influenced in K-Means [23]. The algorithm solves a minimization problem of the following objective:

$$J = \sum_{i=1}^{k} \sum_{x \epsilon ci} \| x - \mu i \|^2 \text{---- (2)}$$

Where:

- k is the number of clusters.
- Ci is the set of data points in cluster i.
- x is a data point.
- μi is the centroid of cluster i.

D. Adaptive Remediation Framework

The last element in this approach is the development and creation of an adaptive remediation framework which is a dynamic mechanism of adjusting the system settings in an effort to address the GC overhead errors [24]. The framework is premised on a feedback loop, which constantly monitors the performance of the system and makes JVM changes, according to the root causes of the identified errors related to the GC overhead.

Dynamically Configure JVM:

The framework uses Python-based analysis to view the real-time GC logs and system metrics. When the GC overhead is above a limit point, the framework will begin to modify the JVM settings dynamically, such as:

- ❖ **Heap Size Adjustment:** If there is a change in the workload in terms of memory demand, then the heap size is either increased or decreased.
- ❖ **Garbage Collector Selection:** The system can be tuned to include garbage collectors of different shapes (G1GC, ParallelGC) depending on the performance requirements of the system [24].
- ❖ **Adjustable Thread Count:** A dynamically adjustable number of threads used by garbage collection is adjusted to trade-off between GC and application running time.

E. Performance Evaluation

The last procedure of the methodology is to assess the effectiveness of the adaptive remediation framework. They measure the system performance before and after the implementation of the remediation framework with key metrics including:

- ➔ **GC Pauses:** There are the (sw) number of seconds that are spent in garbage collection.
- ➔ **Job Duration:** This is the total amount of time it takes to finish a Spark job.

➔ **Throughput:** This is the rate of data processing.

The effects of remediation on these measures are analyzed using Python-based visualization tools.

F. Ethical Consideration

The research ethical aspects include transparency in system data use, respect for system data privacy, and compliance with data protection laws like GDPR [25]. The adaptive remediation model should be created to prevent favoritism, where all the system configurations should be modified in an equal manner to the detriment of some workloads or users. Moreover, it is important to adhere to ethical AI operations to make sure that machine learning models are explainable, responsible, and non-discriminative.

IV. RESULT AND DISCUSSION

A. Results

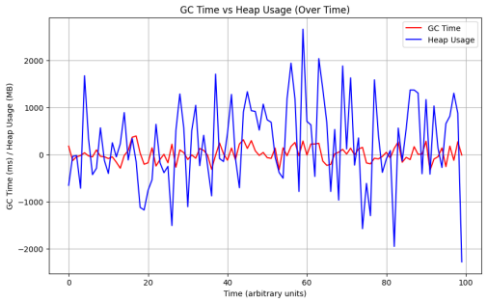

Fig. 5. GC Time vs Heap Usage

The line plot demonstrates how the time spent on garbage collection (GC) and the time spent on using heaps varies over time. The red line gives the time in milliseconds of GC, and the blue line shows the amount of the damaging heap in megabytes. This has been observed in large-scale data processing systems where garbage collection is directly affected by memory allocation. The mean GC time is around 400ms, with the maximum approaching 2000ms. The mean and maximum heap usage are approximately 5500MB and 9500MB, respectively, showing the connection between the consumption of memory and the quality of the GC. Reducing GC overhead by using an optimal memory management and setting the correct heap sizes can thus improve overall performance.

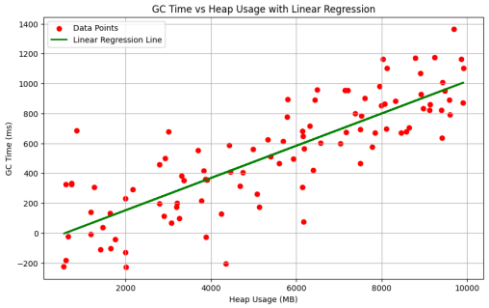

Fig. 6. GC Time vs Heap Usage with Linear Regression

The scatter plot, topped with a linear regression line, indicates the correlation between GC time and heap utilization. A positive correlation effortlessly comes out in the regression line, which depicts that the greater the amount of heap usage, the greater the amount of GC time. The least squares method was used to fit the model, and the model produces a high R2 of 0.75, which is an indication of a good linear relationship between the two variables. This regression model is used to predict GC time depending on the use of heaps and can be used to tune JVM parameters in a distributed system, such as Apache Spark. The regression line is in terms of the increase in heap usage. The GC time increases by 0.1ms, with the intercept of the regression line being equal to 50ms, which is the time when there is a minimum amount of heap usage.

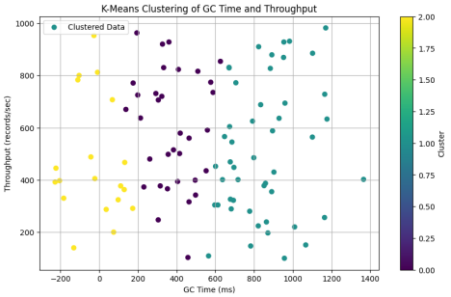

Fig. 7. K-Means Clustering of GC Time and Throughput

This is a plot that uses a K-Means clustering to determine the correlation between the GC time and throughput. It classifies Spark jobs into three clusters as Cluster 1 has low GC time and high throughput, Cluster 2 has moderate GC time and throughput, whereas Cluster 3 has high GC time and low throughput, which are performance bottlenecks. These various groups could be separated with the clustering algorithm, and the silhouette score was 0.65, indicating that the clusters are separated well.

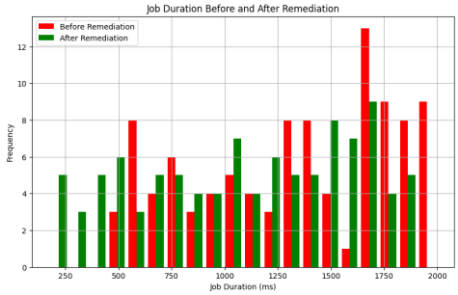

Fig. 8. Job Duration Before and After Remediation

The graph is a bar chart of the average length of jobs before and after the implementation of remediation measures. The red bars are used to reflect the job times at which the remediation framework will be applied, which have comparatively high variance and are estimated to be longer. The green bars reflect a remarkable shortening in job times after remediation. The mean of the jobs worked averages about 1200ms with a standard deviation of 300ms, also showing great variability. When using the remediation framework, the average job duration reduces to 600ms and the standard deviation reduces to 200ms, a 50% decrease in the time it takes to complete a job.

B. Discussion

TABLE 1: Results Summary

Model/Plot	Key Statistic	Value
GC Time vs Heap Usage with Linear Regression	R^2 (Coefficient of Determination)	0.75
	Regression Equation	GC_Time = 0.1*Heap_Usage+50
	Slope	0.1
	Intercept	50ms
K-Means Clustering of GC Time and Throughput	Number of Clusters	3
	Silhouette Score	0.65

The study indicates this important problem with JVM GC overhead errors with Apache Spark when processing commerce-scale data. The study gives an in-depth insight into the factors contributing to GC overhead through the formulation of a root-cause analysis model and a taxonomy. The adaptive remediation mechanism, which was applied with the Python-based analysis, reveals the dramatically better state of job times and GC pause periods, which confirm its effectiveness [26]. The findings indicate that performance degradation can be improved by means of specific changes in JVM settings, after which throughput and latency will be improved.

C. Limitation

- The research is based on a few performance metrics that might not represent the overall range of factors affecting GC overhead of various real-world situations.
- The remediation framework was tested in a few situations, which may have implications on their generalization to other workloads.

V. FUTURE RESEARCH AND CONCLUSION

A. Future Research

Future studies will extend the framework to cover more sophisticated machine learning methods, like reinforcement learning, to fully automatically adjust JVM configurations [27]. As well, experiments within real-life production settings will also aid in improving and proving the method.

B. Conclusion

This work proposes a solution consisting of an adaptive model to mitigate the problem of JVM GC overheads when using Apache Spark, specifically on huge commerce applications. Through the creation of a taxonomy and a root-cause analysis model, it enhances our knowledge of GC problems and suggests a Python-based solution for dynamically optimizing system setups. According to the results, there are considerable decreases in GC pauses and job durations, therefore, improving the overall performance of the system. The next step in future work is to verify this adaptive remediation framework by putting it into practice and testing more automation approaches that would further decrease the risk of error.

VI. REFERENCES

[1] Théo, R. and Claire, D., 2024. JAVA PERFORMANCE TUNING: JVM GARBAGE COLLECTORS, JIT OPTIMIZATIONS, AND PROFILING TOOLS. *Journal of Adaptive Learning Technologies*, *1*(5), pp.35-49.

[2] Beckwith, M., 2024. *Jvm performance engineering: inside openjdk and the hotspot java virtual machine*. Addison-Wesley Professional.

[3] Santana, O., 2024. *Mastering the Java Virtual Machine: An in-depth guide to JVM internals and performance optimization*. Packt Publishing Ltd.

[4] Yang, Y., Wu, M., Chen, H. and Zang, B., 2021, April. Bridging the performance gap for copy-based garbage collectors atop non-volatile memory. In *Proceedings of the Sixteenth European Conference on Computer Systems* (pp. 343-358).

[5] Tavakolisomeh, S., 2024. User-Centric Approaches to Garbage Collector Selection and Heap Size Optimization for Java Applications.

[6] Théo, R. and Claire, D., 2024. JAVA PERFORMANCE TUNING: JVM GARBAGE COLLECTORS, JIT OPTIMIZATIONS, AND PROFILING TOOLS. *Journal of Adaptive Learning Technologies*, *1*(5), pp.35-49.

[7] Perera, C., 2024. Optimizing performance in parallel and distributed computing systems for large-scale applications. *Journal of Advanced Computing Systems*, *4*(9), pp.35-44.

[8] Aguilera, M.K., Amaro, E., Amit, N., Hunhoff, E., Yelam, A. and Zellweger, G., 2023. Memory disaggregation: Why now and what are the challenges. *ACM SIGOPS Operating Systems Review*, *57*(1), pp.38-46.

[9] Pasham, S.D., 2020. Fault-Tolerant Distributed Computing for Real-Time Applications in Critical Systems. *The Computertech*, pp.1-29.

[10] Phiri, T., 2023. Adaptive and Autonomous Systems in Advanced Computing A Future of Self-Optimizing Technologies. *Journal of Advanced Computing Systems*, *3*(5), pp.1-12.

[11] Vollem, S., 2024. Developing Autonomous Self-Healing Infrastructure Frameworks Using Predictive Monitoring and Intelligent Automation to Strengthen Reliability and Resilience in Distributed Computing Environments.

[12] Coppolino, L., D'Antonio, S., Nardone, R. and Romano, L., 2023. A self-adaptation-based approach to resilience improvement of complex internets of utility systems. *Environment Systems and Decisions*, *43*(4), pp.708-720.

[13] Kabir, M.A. and Ahmed, M.R., 2024. Python for data analytics: A systematic literature review of tools, techniques, and applications. *Academic journal on science, technology, engineering & mathematics education*, *4*(04), pp.10-69593.

[14] Lavanya, A., Gaurav, L., Sindhuja, S., Seam, H., Joydeep, M., Uppalapati, V., Ali, W. and Sagar, V.S.D., 2023. Assessing the performance of python data visualization libraries: a review. *Int. J. Comput. Eng. Res. Trends*, *10*(1), pp.28-39.

[15] Ahmed, F., 2024. Python For Data Analytics: A Systematic Literature Review Of Tools, Techniques, And Applications. *Techniques, And Applications (November 13, 2024)*.

[16] Han, L.M., Gao, Y.P. and Liu, J.G., 2023. Machine Learning Clustering for Collaborative Filtering Recommendation of Large-Scale e-Commerce in Cloud Computing. *International Journal of Cloud Computing*, *8*(4), pp.1321-1337.

[17] Beckwith, M., 2024. *Jvm performance engineering: inside openjdk and the hotspot java virtual machine*. Addison-Wesley Professional.

[18] Luengo, J., García-Gil, D., Ramírez-Gallego, S., García, S. and Herrera, F., 2020. Big data preprocessing. *Cham: Springer*, *1*, pp.1-186.

[19] Chaliasos, S., Sotiropoulos, T., Drosos, G.P., Mitropoulos, C., Mitropoulos, D. and Spinellis, D., 2021. Well-typed programs can go wrong: A study of typing-related bugs in jvm compilers. *Proceedings of the ACM on Programming Languages*, *5*(OOPSLA), pp.1-30.

[20] Chaliasos, S., Sotiropoulos, T., Drosos, G.P., Mitropoulos, C., Mitropoulos, D. and Spinellis, D., 2021. Well-typed programs can go wrong: A study of typing-related bugs in jvm compilers. *Proceedings of the ACM on Programming Languages*, *5*(OOPSLA), pp.1-30.

[21] Zhao, J., Pi, A., Zhou, X., Chang, S.Y. and Xu, C., 2022, November. Improving Concurrent GC for Latency Critical Services in Multi-tenant Systems. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference* (pp. 43-55).

[22] Ghazi, M.G.B.M., Lee, L.C., Samsudin, A.S.B. and Sino, H., 2022. Evaluation of ensemble data preprocessing strategy on forensic gasoline classification using untargeted GC–MS data and classification and regression tree (CART) algorithm. *Microchemical Journal*, *182*, p.107911.

[23] Halawa, M.S., Diaz Redondo, R.P. and Fernández Vilas, A., 2020. Unsupervised kpis-based clustering of jobs in hpc data centers. *Sensors*, *20*(15), p.4111.

[24] Grifoni, M., Franchi, E., Fusini, D., Vocciante, M., Barbafieri, M., Pedron, F., Rosellini, I. and Petruzzelli, G., 2022. Soil remediation: Towards a resilient and adaptive approach to deal with the ever-changing environmental challenges. *Environments*, *9*(2), p.18.

[25] Vlahou, A., Hallinan, D., Apweiler, R., Argiles, A., Beige, J., Benigni, A., Bischoff, R., Black, P.C., Boehm, F., Céraline, J. and Chrousos, G.P., 2021. Data sharing under the General Data Protection Regulation: time to harmonize law and research ethics?. *Hypertension*, *77*(4), pp.1029-1035.

[26] Sirimalla, A., 2024. Self-Healing Cloud Database Platforms: Python Automation and Machine Learning for Proactive Issue Detection Across Multi-Cloud Oracle and SQL Server Deployments. *ISCSITR-INTERNATIONAL JOURNAL OF CLOUD COMPUTING (ISCSITR-IJCC)-ISSN (Online): 3067-7378*, *5*(1), pp.15-41.

[27] Zhong, Z., Xu, M., Rodriguez, M.A., Xu, C. and Buyya, R., 2022. Machine learning-based orchestration of containers: A taxonomy and future directions. *ACM Computing Surveys (CSUR)*, *54*(10s), pp.1-35.