International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

Optimized Isolation Forest Based Docker Container Anomaly Monitoring System

Ms. Mohuya Pal
Department of Computer Engineering
JSPM's Rajarshi Shahu College of Engineering
Pune, India
mahuapal.edu@gmail.com

Dr. Prema Sahane
Department of Computer Engineering
JSPM's Rajarshi Shahu College of Engineering
Pune, India
pbsahane_comp@jspmrscoe.edu.in

***Abstract*— This paper presents an enhanced anomaly detection system for Docker containers, leveraging an improved version of the Isolation Forest algorithm. Due to the highly dynamic and ephemeral nature of containerized environments, traditional monitoring approaches often fall short in identifying issues such as performance degradation, security breaches, and inefficient resource usage. Docker containers are widely adopted for deploying scalable applications in isolated environments, but their complexity makes consistent and accurate monitoring a challenge. To address this, our approach integrates hyperparameter tuning and optimized feature selection into the Isolation Forest model, significantly improving its ability to detect abnormal behavior. We evaluate the system using real-world Docker container data, and the results demonstrate superior performance compared to conventional methods. The proposed solution effectively identifies both known and previously unseen anomalies in real-time, without relying on labeled datasets.**

Keywords*— Anomaly monitoring, Docker container, Log analysis, Isolation forest.

I. Introduction

With the growing accessibility of cloud computing, more companies are building their own data centers to better serve their customers' diverse needs. Docker containers have gained popularity in this space because of their quick deployment and startup times. As virtualization has become a core element of modern data centers, containers like Docker are now widely used by industry leaders including Amazon, IBM, and Oracle.

Despite the increasing adoption of container clusters, concerns about their security and stability continue to grow. A notable example is a major outage in Amazon's cloud services—an event heavily reliant on containerized and virtualized infrastructure—that caused widespread disruption across hundreds of websites and apps. Such incidents underline the critical need for timely and accurate anomaly detection in containers to ensure the reliability of cloud services.

Monitoring a multitude of resources in a highly dynamic environment presents a major challenge, especially when aiming to minimize infrastructure overhead. Rule-based anomaly detection methods use preset thresholds for different metrics. These systems often assume that each host runs a single container and adjust thresholds as new containers are added. However, this model becomes impractical as the number and variety of containers scale up. Traditional statistical methods, which typically rely on univariate data assumptions, struggle to handle the complexity of highdimensional data and often fail to detect subtle anomalies. To address these issues, researchers have introduced densitybased techniques such as Local Outlier Factor (LOF) and Angle-Based Outlier Detection (ABOD), which identify anomalies by analyzing local data density or angular variance. However, these methods can become computationally expensive for large-scale datasets.

Tools like Ganglia, Nagios, Akshay, and cAdvisor use fixed monitoring intervals to track system behavior. While short intervals can quickly catch anomalies, they significantly increase system overhead. Longer intervals reduce this burden but delay detection. Therefore, it's vital to adjust monitoring intervals dynamically based on the system's real-time state.

Anomalies in containers are often linked to unusual resource usage patterns. For instance, excessive CPU usage might indicate an infinite loop, while increasing memory consumption could signal a memory leak. Monitoring resource usage is thus key to early anomaly detection.

This study proposes an anomaly monitoring system tailored for containers, utilizing an optimized Isolation Forest algorithm. The system passively collects resource usage metrics and calculates anomaly scores while taking into account the unique workload characteristics of each container application. Resource metrics are weighted based on their importance to the container's function—for example, I/Ointensive applications would prioritize disk read/write

International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

rates. This weighted feature approach improves detection accuracy.

When an anomaly score exceeds a predefined threshold, the system investigates container logs to identify the root cause. It also adapts the monitoring frequency depending on the severity of the issue, helping to reduce both response times and unnecessary monitoring overhead.

The most important contributions that this work makes are as follows:

- Developing a system to detect and track anomalies in Docker containers; this system will keep an eye on multidimensional resource metrics, tweak monitoring intervals automatically, and look into what causes them.

Improved anomaly detection based on container application workloads is possible with the help of this article's new iteration of the Isolation Forest method, which accounts for resource metric weights.

- Using both simulated and actual AWS settings, we will test the system and method for a variety of unique circumstances to see how well they handle detection accuracy, surveillance latency, and log analysis.

II. LITERATURE SURVEY

Docker is a lightweight virtualization technology operating as an operation running on the host computer, separating its resources via kernel-level namespaces, enabling processes in containers and the host to communicate without interference. Unlike virtual machines (VMs), which employ hardware-layer virtualization, Docker employs operating system-level virtualization, resulting in higher performance and efficiency [14]. Let R denote the performance efficiency; Docker achieves RD > RVM, where RD and RVM represent the resource utilization efficiency of Docker and VMs, respectively. Docker eliminates hardware emulation and additional OS layers required by VMs [15]. Given L as the layers of abstraction, Docker achieves LD < LVM, reducing computational overhead. Consequently, startup time T is minimized, with TD < TVM, typically a few seconds compared to several minutes for VMs. Additionally, Docker exhibits mobility and scalability, operating across various platforms, denoted as P, with PD > PVM These properties have led researchers to adopt Docker in diverse domains. For example, multi-task PaaS cloud infrastructure was deployed using Docker by Tihfon et al. [16], achieving rapid application deployment and optimization. Nguyen et al. [18] used Docker to streamline MPI clustering for HPC, drastically cutting down on setup time. Clusters of autonomously expanding networks were refined by Julian et al. [19]. , demonstrating the scalability of Docker in large production environments.

The Isolation Forest algorithm (iForest) identifies anomalies without requiring a predefined mathematical model or training. It partitions data based on random selection of features and thresholds, building isolation trees (iTree). Consider a dataset with NNN data points,{x1,x2,…,xN}, where xi represents individual data. An iTree is constructed by selecting n samples (n<N) and recursively dividing data using a feature f and threshold p. The process terminates when the tree height equals log2(n) or samples cannot be divided further.

The path length h(x)h(x)h(x distance from data point x to the root node is a measure of how sparse x is. The discrepancy of x in any number of samples may be quantified using anomaly scoring, s(x,n):

$$s(x,n) = 2^{\left(-\frac{E(h(x))}{c(n)}\right)} \quad (1)$$

$$c(n) = 2H(n-1) - (2(n-1)/n), H(k) = ln(k) + \xi \quad (2)$$

with γ as Euler's constant. Values of s(x,n) close to 1 indicate high anomaly probability, while values near 0 suggest normalcy.

Monitoring systems have evolved significantly to address handle resources in contexts that use containers. Gmond, gmetad, and a website front-end and back are the building blocks of Ganglia, an open-source solution for cluster monitoring. to collect and visualize data [6]. However, its inability to analyze anomalies limits its applicability. Nagios [8] extends monitoring capabilities by offering exception notifications but relies on static thresholds unsuitable for dynamic container scenarios. Let T denote the threshold; static T values lead to misclassification in dynamic systems where ΔT =0

Using Docker, Anand et al. [12] suggested a way to monitor containers.

APIs to estimate the standard deviation σ\sigmaσ of resource usage. Data storage is conditional upon σ>σth, where σth is a predefined threshold. Despite storage efficiency, the absence of alarm functionality limits its utility. Google's cAdvisor [13] collects and aggregates container metrics but is constrained to single-node environments, lacks data storage, and provides only a one-minute sliding window.

Anomaly detection methods rely on statistical, entropybased, distance-based, and density-based approaches. Statistical methods [9] construct models assuming a standard distribution. For multidimensional metrics, these models struggle with accuracy due to noise ϵ\epsilonϵ, where ϵ\epsilonϵ disrupts the model. Entropy-based methods [22] compare entropy fluctuations over time, identifying anomalies when ΔS>δ is entropy change, and δ is a predefined threshold. However, these methods are unsuitable for dynamic container environments.

International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

Distance-based methods [23] calculate inter-data distances D(xi,xj). Data xix_ixi is anomalous if the number of neighbours N(xi)<p, where p is the neighbor threshold. This method fails in multi-cluster scenarios, as clusters of anomalous data Ca may resemble normal data clusters Cn. Methods that rely on density, like the Locally Outlier Score (LOF) [10], calculate a score based on density:

$$\mathrm{LOF}(x) = \frac{\mathrm{density}(x)}{\mathrm{density}(\mathrm{neighbors}(x))}.$$

High LOF values indicate anomalies, but computational overhead limits its scalability for large datasets $|X| \gg 0$.

While these methods exhibit strengths, their limitations highlight the need for optimized approaches. Advanced systems must integrate scalable algorithms, adaptive thresholds, and efficient anomaly scoring to address the dynamic and multidimensional nature of containerized environments.

III. METHODOLOGY

A. Architecture

Four main components—the The four main components of the architecture are the monitoring agent, the monitoring data storage, the oddity detection, and the anomaly analysis.

Monitoring system shown in Figure 1. Every host machine has a single monitoring agent that uses a non-invasive technique to record container resource use rates. Collecting monitoring data from every host machine, the Monitoring Data Storage module keeps just the most recent period of data. The Anomaly Detection module receives this formatted data then for additional handling.

The Anomaly Detection module searches the acquired data for anomalies using an iForest-based evaluation technique. The Anomaly Analysis module receives then detected aberrant container information. Retrieving the logs of the impacted containers from their respective hosts, this module examines the logs and finds the root cause of the abnormalities.

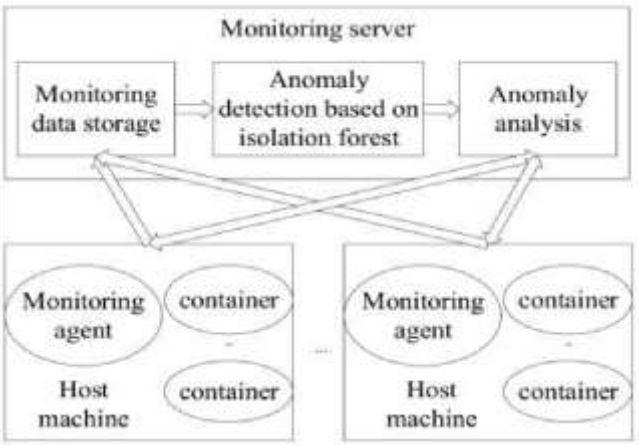


Fig. 1. System architecture

B. Monitoring agent

Several functional modules make up the monitoring agent' internal structure, as shown in Figure 2: monitoring data collecting, monitoring data processing, container information management, monitoring period adjustment, data collecting control, log collecting, and transmission.

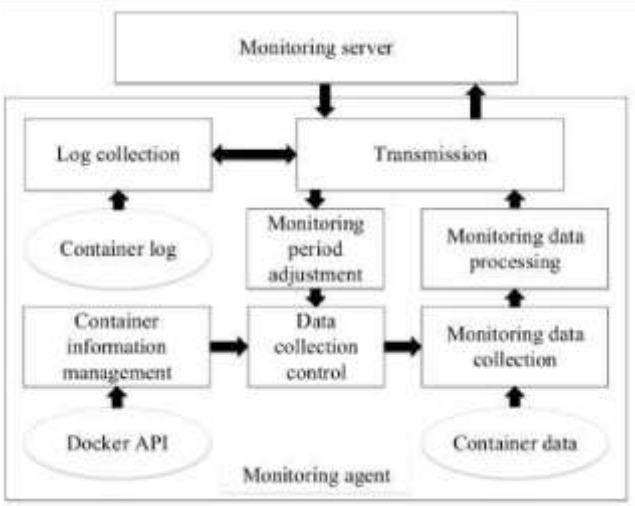


Fig. 2. The internal design of the monitoring agent

Keeping an Eye on Data Gathering

Real-time data collecting from every running container on the host machine is handled in this module. Typical measurements comprise container ID, timestamp, CPU use, memory use, disc I/O rates, and network speed.

Data Processing for Monitoring

After the data is collected, this module does two phases. First, the data is organized and presented in a way that is compatible with databases; this ensures that it captures

containers ID, date, and resource utilization. Second, it does data compilation for mirrored containers that are executing the same service.

Container Information Management

This module keeps tabs on the current status of container using the Docker API. It records events such as containers starting or stopping, IDs, details of tasks, and information about mirroring. After that, this data is sent to the Data Acquisition Control module.

Monitoring Period Adjustment

This module uses the Docker API to monitor the container's current condition. Events such containers beginning or stopping, IDs, task descriptions, and mirroring information are recorded. The Data Acquisition Controller module then receives this data.

Data Collection Control

Acting as the monitoring agent' central control unit, this module preserves a queue of containers to be watched over. It determines which container to utilize for future data collection based on the last collecting duration and the selected monitoring period. In order to add or remove vessels from the queue, it also integrates information on vessel lifetimes from the container-specific Information Management database. Cutting container monitoring times in half for those identified as potentially aberrant gives the module dynamic control over the monitoring sequence. On the other hand, containers judged stable deprioritizes in the queue and have twice as monitoring periods.

Log Collection

This module gathers logs for designated containers under direction from the monitoring server, formats them suitably, and forwards them for transmission.

Transmission

This module carries two different obligations. It runs the commands to the pertinent modules and manages command reception from the monitoring server. It presents gathered monitoring data to the server at the same time.

Coordinating these modules helps the monitoring agent to guarantee effective and flexible resource monitoring, so supporting quick anomaly detection and resolution.

C. Monitoring data storage

As soon as the surveillance agent has collected data, its data storage module is responsible for sending it to the anomaly recognition module in the specified format. An free to download distributed time-series database designed for event and metric data, InfluxDB [26] stores the container information. The JSON format is supported by InfluxDB, allowing both the surveillance agent and the finding anomalies module to communicate with data seamlessly.

Data about containers is kept in a specific table in InfluxDB. This database contains the container ID, as well as information on the CPU, memory, disk, network, and the time of data collection. The database only keeps the most recent hour's worth of monitoring data in order to minimize storage overhead.

There is also a storage controls table that contains the following information: container ID, latest update time, and the number of rows in the data table. There are three primary uses for this table in container information management:

Creation and Insertion

Data is entered into the data table as it is received from the monitoring agent. The absence of an existing container ID in the data table denotes that the data is associated with a newly begun container. An entry is generated in the storage control table by the database for the new container in this instance. For containers where the ID already exists, updating the storage control table with the amount of rows and the time of modification is done.

Deletion

The scanning of the storage control table occurs every ten minutes. Assumption of closure occurs when more than ten minutes pass without an update to a container's details. Consequently, the container’s data is deleted from both the data table and the storage control table.

Data Transmission to Anomaly Detection Module

Since the anomaly detection module requires a sufficient volume of data to build an Isolation Forest model, the monitoring data storage module sends data to the anomaly detection module when the number of rows in the storage control table for a container reaches 100. At this point, the corresponding 100 rows of data from the container’s data table are transmitted in JSON format to the anomaly detection module for processing.This structured approach ensures efficient data management, minimal storage use, and timely data transmission to support anomaly detection.

D. Anomaly detection

a. Data cleaning

Data loss, repetition, or manipulation during transfer and retention is a real possibility with container data due to its massive volume. Eliminating duplicate entries and dealing with missing data is a necessary first step before building an isolation forest. To preserve the forest's structure, unnecessary data must be removed. When a greater proportion of lines are missing or when five or more points in row are gone, it becomes troublesome due to missing data. In the event of a

International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

catastrophic data loss, the anomaly detection method will not include the impacted period.

b. Optimization of Isolation Forest Algorithm

Introduction and Resource Weight Calculation:

The classic iForest algorithm struggles with container monitoring due to equal weighting of resource indicators (CPU, memory, disk, and network speed). Containers with different resource dependencies (e.g., CPU-intensive or IOintensive) may be inaccurately monitored using the same standard. This work presents an optimization strategy to enhance accuracy. The method entails assigning weights to each resource indicator, which increases the likelihood of selecting relevant indicators for anomaly detection. In order to maximize monitoring, a self-learning method determines bias parameters (M) for every resource while the container is in use. Formula (3) shows the normal container consumption for each resource and how the bias parameters (M) are derived. This leads to the introduction of a self-learning mechanism for optimizing resource bias.

$$M = \begin{cases} 0, & (\sum_{i=1}^{p} N_i = 0) \\ W_0 + \frac{\sum_{i=1}^{p} f(N_i - \epsilon)}{p} \end{cases} \quad (3)$$

With a starting value of 1, W_0 denotes the resource metric's initial weight. At time i, Ni represents the rate of resource utilization, and ε stands for the resource threshold. The value of p represents the frequency of the resource's measurement. This function's value is 1 when x is greater than 0, and 0 otherwise. The resource's weight is set to zero if its value is always zero, which indicates that the container does not use that resource. The bias toward that resource in containers increases as the M value rises.

Every resource metric is given a weight by the bias parameter M. The starting weight for all resources is 1. Every ten minutes, the weight is changed from the default to M dependent on the data usage rate for that period. This feature value selection process concludes with a weighted random selection technique. Algorithm 1 displays the algorithm's pseudocode.

Three, two, and four are the resource weights. Mall is the sum of all weights. As a feature to partition the data collection, the resource's index number is last returned as i. Data from 0 to Mall is represented by R.

abnormality values for multidimensional resource metrics are computed using the iForest algorithm, however the underlying measure that generates the abnormality cannot be identified. It could be difficult to determine the precise source of a spike in CPU or memory utilization because the anomalous value might be comparable. This paper offers a solution by outlining a procedure to identify the unusual resource metric.

Algorithm 1 Weighted random algorithm

Input: M_1, M_2, M_3, M_4 ///M_1 is CPU weight, M_2 is Memory weight, M_3 is IO weight, M_4 is Network weight.

Output: i ///A feature among the four features (CPU usage rate, Memory usage rate, IO rate, Network usage rate).

1: $M_{all} = M_1 + M_2 + M_3 + M_4$
2: $R = Random() * M_{all}$
3: **for** $i = 4, R > 0, i = i - 1$ **do**
4: $R = R - M_i$
5: **endfor**

The feature that was used to divide the data at that point is regarded the isolation feature for that data when constructing an isolation tree. This feature indicates which feature is responsible for isolating the data.

Each data point is assigned an isolation feature group, where each feature's usage count is tracked across the isolation trees. The more frequently a resource metric appears in the isolation feature group, the more likely it is the cause of the anomaly. This method is based on the premise that data with feature values that significantly differ from others are more likely to be isolated, thus the isolation feature is often the one with the highest anomaly value.

When a container is deemed anomalous, its isolation feature group is compared with that of normal data. The ratio of corresponding values in the groups is calculated, and a higher ratio indicates a higher degree of anomaly for the metric.

A number of iTrees, or random binary trees, come together to form an isolation forest. Every node can be either a parent node with two children or a leaf node, the latter of which represents data in isolation.

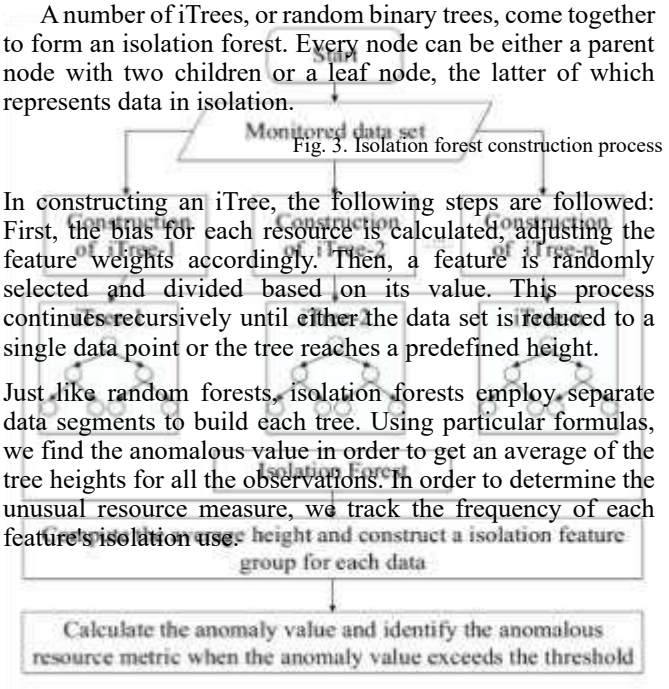


Fig. 3. Isolation forest construction process

In constructing an iTree, the following steps are followed: First, the bias for each resource is calculated, adjusting the feature weights accordingly. Then, a feature is randomly selected and divided based on its value. This process continues recursively until either the data set is reduced to a single data point or the tree reaches a predefined height.

Just like random forests, isolation forests employ separate data segments to build each tree. Using particular formulas, we find the anomalous value in order to get an average of the tree heights for all the observations. In order to determine the unusual resource measure, we track the frequency of each feature's isolation use.

International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

E. Monitoring period adjustment

To enhance monitoring timeliness, the monitoring period can be shortened to gather more data, allowing earlier detection of changes in anomaly values. To establish the probability of an anomaly, one sets an anomaly sensitivity threshold, abc, has the following relationship to the anomaly identification threshold, d:

$$f = \frac{d + p}{2} \qquad (4)$$

The default value for the normal anomaly, p, is 0.5. We will closely watch the container if the average anomaly value is found to be between ff and dd, which indicates a possible concern. The intensive type message is sent by the monitoring server with the container id 100. The time is cut in half for close observation. A message of type "extensive" is sent whenever the anomalous value falls below ff, signaling that the monitoring period should be reset to its initial value.

F. Anomaly Analysis

After the detection module finds an unusual container, the problem analysis module looks through its records to find out why. Its two primary parts are the log data collecting module of the monitoring agent and the system that utilizes that data.

A. Log Preprocessing

The first stage before analysis is log preprocessing, which aims to reduce storage and analytical cost by removing useful log events.

Container system logs are saved in JSON format, often containing unnecessary escape sequences (e.g., u0008), which increase log volume. These sequences are filtered out during preprocessing. Additionally, irrelevant application logs, like web access logs, are removed as they don't contribute to anomaly analysis. This is done using regular expressions in the Logstash configuration file [27] to filter and drop unnecessary log content, with the remaining data stored in the database.

B. Log Analysis

The log research module can correctly update the database by comparing pre-processed record incidents with a rule database and identifying similar itemsets. Both "normal" rules, used for typical container operations, and "exception" rules, based on empirical and historical data, make up the rule database.

The analysis process follows these steps: 1) Match logs with empirical exception rules; if successful, trigger an alarm. If not, use the Apriori algorithm to mine frequent itemsets. 2) Match mined itemsets with normal or historical exception rules and trigger alarms as needed. 3) If no match is found, the administrator adds the itemsets to the rule databases [28].

IV. RESULTS ANALYSIS

Experiments were carried out in both simulated and realworld cloud environments. In the simulated setup, one machine hosted the monitoring server, while another ran the monitoring agent and Docker containers. The real-world environment utilized Amazon EC2 instances with two configurations: t3.medium, offering two CPU cores and 4 GB of RAM, and t3.small, a free-tier instance with one CPU core and 2 GB of RAM. Both setups used Ubuntu 16.04 and Docker 18.03.1-ce. All monitoring components were deployed within the cloud infrastructure.

Two representative applications were chosen to test the monitoring system: Memcached and Web Search from CloudSuite. Memcached is a distributed memory object caching system aimed at reducing database load in dynamic web applications. The Web Search workload, built on Apache Solr, uses a 12 GB search index created via Apache Nutch. To simulate real-world demand, Mutilate was used to generate load on Memcached, while Faban was used for Web Search.

Since there is no standardized benchmark for containerlevel anomalies, the study defined four common fault types: CPU overload, memory leak, disk I/O overload, and network congestion. CPU anomalies were introduced by injecting a stress loop that used 100% of the processor. Memory leaks were created by allocating heap memory without releasing it. Disk faults were simulated using FIO to generate excess read/write operations, and network congestion was simulated by limiting bandwidth with wondershaper.

To assess detection performance, both the detection rate and false alarm rate were measured. The optimized isolation forest (iForest) algorithm demonstrated higher accuracy and lower false alarms compared to the original iForest and the Local Outlier Factor (LOF) algorithm. Especially for Memcached, where resource usage is more stable, optimized iForest provided better results. In contrast, Web Search exhibited higher variability under normal conditions, which led to more false positives, particularly for LOF.

In practical scenarios, malicious processes might not consume all CPU resources but still degrade performance. For instance, a simulated web attack using siege consumed 60–80% of CPU. In such cases, optimized iForest consistently outperformed the original, which struggled to identify anomalies at lower resource utilization. Although the original iForest showed no false alarms, its detection capabilities were inadequate under subtle attack conditions. The optimized version maintained strong detection with a tolerable increase in false alarms, especially in single-core environments where resource fluctuations were more pronounced.

International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

An example anomaly detection case was conducted on the Memcached container. Three events were introduced: CPU overuse, network congestion, and a legitimate workload increase. The optimized iForest successfully flagged the first two as anomalies due to significant deviations in monitored metrics. However, the workload increase, which did not disrupt normal behavior, was correctly not flagged. This demonstrates the system's ability to differentiate between normal fluctuations and real faults.

After detecting an anomaly, the system identifies the most likely affected metric by comparing the frequency of selected features during anomalous versus normal periods. Metrics with notably higher isolation feature ratios are deemed the source of the issue, enabling accurate fault localization.

To determine an appropriate anomaly detection threshold, multiple tests with varying thresholds were performed. As the threshold increased, both detection and false alarm rates declined. The optimal setting balances sensitivity and accuracy.

The number of isolation trees in iForest significantly impacts performance. Detection improved and false alarms decreased as the number of trees increased, but after reaching 100 trees, additional gains diminished while computation time rose. Thus, 100 trees was selected as the optimal configuration.

Monitoring delay—the time between fault injection and detection—was evaluated under two conditions: fixed and adaptive monitoring intervals. Using a fixed 4-second interval led to delays between 40 and 55 seconds, due to the time required to collect sufficient data to rebuild the model. In contrast, a dynamically adjusted monitoring interval reduced detection times by an average of 13.5%. When anomalies were detected, the monitoring frequency increased temporarily to capture more granular data, accelerating detection. Once the container returned to normal, the interval reset to its original value.

The system includes a log analysis component for postanomaly investigation. Two attack scenarios were tested. In the first, a disk attack simulated by continuous file operations caused abnormal disk usage. After detection, system logs were filtered, reducing their size significantly and revealing repeated file creation and deletion. In the second case, a network attack was simulated by flooding a website hosted in a container with GET requests. Again, the logs were trimmed and analyzed, revealing over 100 requests from a single IP address, identifying it as the anomaly source.

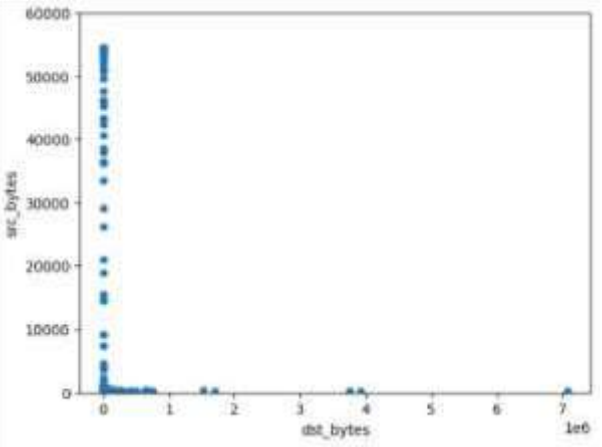
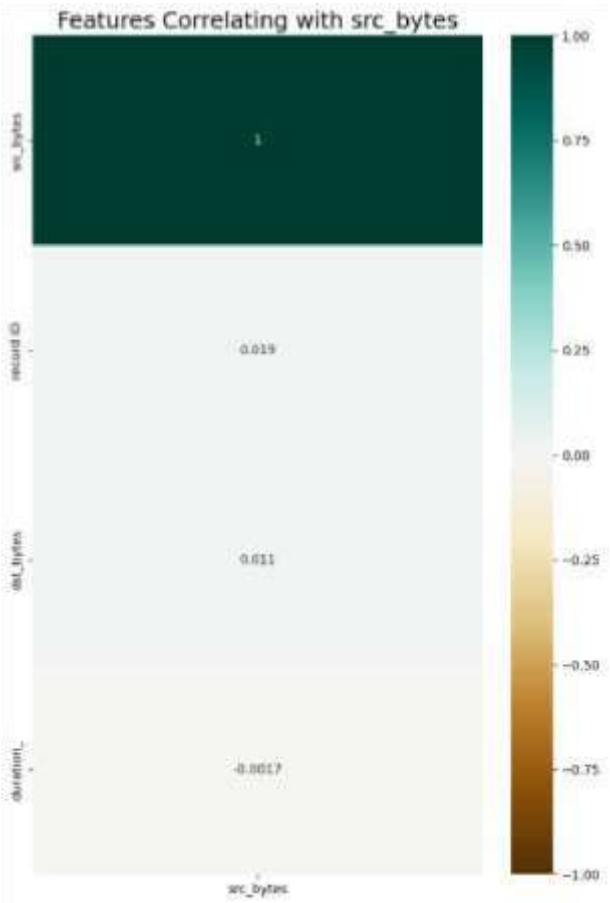


Fig. 4. Scatter Plot of src_bytes vs dst_bytes

Figure 4 shows that as scatter plot displays the relationship between source and destination bytes in network traffic data. Most points cluster around low dst_bytes values, with a few high outliers. The dense vertical line suggests frequent small responses to larger incoming data — a common pattern in anomaly detection for network monitoring.



International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

Fig. 5. Scatter Plot of src_bytes vs dst_bytes

Figure 5 shows that as heatmap highlights the correlation of src_bytes with other features. All correlations are very weak, with values near zero, indicating low linear dependency. Such low correlation implies each feature adds unique information, which is beneficial in detecting anomalies across multiple dimensions.

Fig. 6. Isolation Forest Model Initialization

Figure 6 shows that as code calculates the estimated contamination — the proportion of suspected outliers — based on src_bytes. The model is trained using the Isolation Forest algorithm with 50 estimators. It helps in identifying anomalous network records by isolating outliers in the dataset.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	255648
1	0.05	0.01	0.02	1022
accuracy			0.99	256670
macro avg	0.52	0.51	0.51	256670
weighted avg	0.99	0.99	0.99	256670

TN: 255372
FP: 276
FN: 1008
TP: 14
TP+FN: 1022
TN+FP: 255648
Accuracy: 0.9949974675653563

Fig. 7. Model Performance Metrics

Figure 7 shows that as model achieves high accuracy (99%), but performs poorly on the minority (anomalous) class, with an F1-score of just 0.02. This indicates class imbalance — the model identifies normal traffic well but struggles to detect anomalies, highlighting the need for improved detection methods or balanced data.

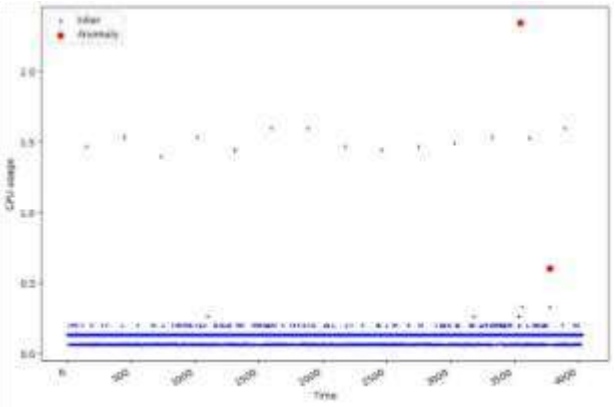


Fig. 8. CPU Usage Anomaly Detection

Figure 8 shows that as plot visualizes CPU usage over time, with normal behavior (inliers) marked in blue and anomalies in red. While most data points show stable usage, the few red points signal sudden spikes that are flagged as anomalous, indicating potential system irregularities.

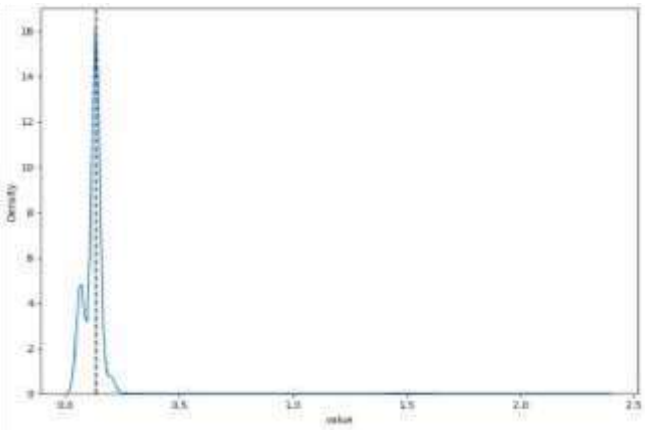


Fig. 9. Distribution of Anomaly Scores from Isolation Forest

Figure 9 shows that as the kernel density estimate (KDE) of anomaly scores generated by the Isolation Forest model. The density curve illustrates how frequently certain scores appear in the dataset. The vertical dashed line represents the estimated contamination threshold — the proportion of data points expected to be anomalous. Values to the right of this threshold are considered potential anomalies. This visualization helps in understanding the separation between normal and anomalous data points based on their anomaly scores.

V. CONCLUSION

This paper presents an online anomaly detection approach for containers that leverages an enhanced Isolation Forest algorithm to analyze multidimensional resource data. By assigning tailored weights to different resource metrics and

adjusting feature selection to reflect the specific behavior of containerized applications, the method improves detection precision. The system also adapts the monitoring interval based on the severity of anomalies, helping to minimize response delays. Furthermore, it incorporates log analysis to pinpoint the underlying causes of unusual behavior. Tests conducted on both virtual and physical cloud environments confirm that the proposed method effectively detects abnormal container activity without degrading system performance.

REFERENCES

[1] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rup- precht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt, "Improving docker registry design based on production workload analysis," in 16th USENIX Conference on File and Storage Technologies (FAST 18). Oakland, CA: USENIX Associa- tion, 2018, pp. 265–278. [2] "Amazons container strategy examined," https://www. informationweek.com/cloud/infrastructure- as- a- service /amazons- container- strategy- examined/a/d- id/1317515.

[3] "IBM containers on bluemix," https://www.ibm.com/bl ogs/bluemix/2015/06/ibm- containers- on- bluemix/.

[4] "Munz docker occs," http://www.oracle.com/technetwo rk/articles/cloudcomp/munz- docker- occs- 3585210.html.

[5] "Amazons cloud service partial outage affects certain websites," http://www.dailymail.co.uk/sciencetech/arti cle- 4268850/Amazons- cloud- service- partial- outage- affects- certain- websites.html. [6] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," Parallel Computing, vol. 30, no. 7, pp. 817– 840, 2004.

[7] "Introduction to Zabbix," http://tim.kehres.com/.

[8] "Nagios: The industry standard in it infrastructure monitoring," https://hops://www.n- agios.org/.

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

[10] M. M. Breunig, H. P. Kriegel, and R. T. Ng, "LOF: iden-tifying density-based local outliers," in ACM SIGMOD International Conference on Management of Data, 2000, pp. 93–104.

[11] H. P. Kriegel, M. S. Hubert, and A. Zimek, "Angle- based outlier detection in high-dimensional data," in ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2008, pp. 444–452.

[12] S. A. K, J. A. K, and K. A, "Resource monitoring of docker containers," International Journal of Advance Engineering and Research Development, vol. 3, no. 2, pp. 146–149, 2016.

[13] "Google.cAdvisor," https://github.com/google/cadvis or. [14] N. Naik, "Building a virtual system of systems using docker swarm in multiple clouds," in IEEE International Symposium on Systems Engineering, 2016, pp. 1–3.

[15] S. Mcdaniel, S. Herbein, and M. Taufer, "A two-tiered approach to I/O quality of service in docker containers," IEEE/ACM Transactions on Networking, vol. 6, no. 1, pp. 42– 55, 2015.

[16] G. M. Tihfon, J. Kim, and K. J. Kim, A New Virtualized Environment for Application Deployment Based on Docker and AWS. Springer Singapore, 2016.

[17] R. Liu, R. Liu, R. Liu, A. C. Arpaci-Dusseau, and R. H. ArpaciDusseau, "Slacker: fast distribution with lazy dock- er containers," in Usenix Conference on File and Storage Technologies, 2016, pp. 181– 195.

[18] N. Nguyen and D. Bein, "Distributed MPI cluster with docker swarm mode," in Computing and Communication Workshop and Conference, 2017, pp. 1–7.

[19] S. Julian, M. Shuey, and S. Cook, "Containers in research: Initial experiences with lightweight infrastructure," in XSEDE16 Conference on Diversity, Big Data, and Science at Scale, 2016, p. 25.

[20] F. T. Liu, M. T. Kai, and Z. H. Zhou, "Isolation forest," in Eighth IEEE International Conference on Data Mining, 2009, pp. 413–422.

[21] N. L. D. Khoa and S. Chawla, Robust Outlier Detection Using Commute Time and Eigenspace Embedding. Springer Berlin Heidelberg, 2010.

[22] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward, "Efficient fault detection and diagnosis in complex software systems with information-theoretic monitoring," IEEE Transactions on Dependable and Secure Computing, vol. 8, no. 4, pp. 510–522, 2011. [23] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algo- rithms for mining outliers from large data sets," in ACM SIGMOD International Conference on Management of Data, 2000, pp. 427–438.

[24] F. Angiulli, S. Basta, and C. Pizzuti, "Distance-based de- tection and prediction of outliers," IEEE Transactions on Knowledge and Data Engineering, pp. 145–160, 2006.