A Docker Container Anomaly Monitoring System Based on Optimized Isolation Forest

Zhuping Zou, Yulai Xie, *Member, IEEE*, Kai Huang, Gongming Xu, Dan Feng, *Member, IEEE*, and Darrell Long, *Fellow, IEEE*

Abstract—Container-based virtualization has gradually become a main solution in today‘s cloud computing environments. Detecting and analyzing anomaly in containers present a major challenge for cloud vendors and users. This paper proposes an online container anomaly detection system by monitoring and analyzing multidimensional resource metrics of the containers based on the optimized isolation forest algorithm. To improve the detection accuracy, it assigns each resource metric a weight and changes the random feature selection in the isolation forest algorithm to the weighted feature selection according to the resource bias of the container. In addition, it can identify abnormal resource metrics and automatically adjust the monitoring period to reduce the monitoring delay and system overhead. Moreover, it can locate the cause of the anomalies via analyzing and exploring the container log. The experimental results demonstrate the performance and efficiency of the system on detecting the typical anomalies in containers in both simulated and real cloud environments.

Index Terms—Docker container, anomaly monitoring, isolation forest, log analysis

1 Introduction

With the popularity of cloud computing platforms, more and more enterprises have their own data centers, providing services to customers with different needs. One of the key technologies in the data center is virtualization. The docker container [1], as a new virtualization technology, has many attractive advantages such as easy to deploy and fast start-up. Thus it has quickly become the darling of major companies (e.g., Amazon [2], IBM [3] and Oracle [4]).

However, with the increasingly large-scale application of container clusters, the issue of container security and stability has also drawn an increasing attention. For instance, the collapse of Amazon Cloud that builds upon container and virtual machine cluster led to invalidation of thousands of websites and apps [5]. Therefore, it is crucial to detect abnormalities in the container in a timely manner to ensure the service quality of the cloud.

As the containers continue to rise and fall, one of the challenges is how to monitor multiple resources at the same time in a dynamic environment with a low overhead. Rule-based methods [6], [7], [8] detect abnormalities by setting a threshold for each metric. They assume that only one container is running on the host at the beginning, and set a fixed threshold for each resource metric of the container. When another container is created with a resource priority, the original resource threshold of the first container is adjusted according to the resource usage of the second container. This adjustment becomes impractical when there exist numerous and dynamically changing containers. The statistics-based method [9] assumes that the data obeys some standard distribution models and finds outliers that deviate from the distribution. Since most models are based on univariate assumptions, they are not applicable to multidimensional data. In order to solve the above-mentioned problems, the academic community has proposed a density-based method such as Local Outlier Factor (LOF) [10] and Angle-Based Outlier Detection (ABOD) [11]. They identify outliers by estimating the density of local data or calculating the angle change. However, they both incur a large computation overhead when the sample data size is large.

The existing monitoring systems (e.g., Ganglia [6], Nagios [8], Akshay [12], cAdviosr [13]) generally adopt a fixed monitoring period to query the abnormality of the system. When the monitoring period is very small, the monitoring system can quickly locate abnormalities. However, this results in a huge system overhead when there are too many monitoring objects. When the monitoring period is large, the monitoring delay will also increase. Thus, it is necessary to adopt a proper monitoring period according to the system running state.

When an exception occurs in a container, it usually causes a change in the resource usage of the container. For example, an endless loop in a running program can eat all the CPU resource, and a memory leak will cause the memory usage to become higher. Therefore, it is necessary to identify the

• *Z. Zou, Y. Xie, K. Huang, G. Xu, and D. Feng are with the School of Computer, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, P.R. China. E-mail: {zouzhup, xugongming38}@gmail.com, {ylxie, keithkhuang, dfeng}@hust.edu.cn.*
• *D. Long is with Jack Baskin School of Engineering, University of California, Santa Cruz, CA 95064. E-mail: darrell@ucsc.edu.*

*Manuscript received 19 Aug. 2018; revised 24 Sept. 2018; accepted 11 Aug. 2019. Date of publication 0 . 0000; date of current version 0 . 0000. (Corresponding author: Yulai Xie.)
Recommended for acceptance by M. Qiu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCC.2019.2935724*

anomaly by monitoring the container resource metrics. This paper proposes a container anomaly monitoring system based on optimized isolation forest. The system first obtains each resource usage rate of each container on the host machine in a non-intrusive manner. When enough monitoring data is collected, the anomaly value of each monitoring data is calculated by using the optimized isolation forest, which takes into account the characteristics of container application workload. Specifically, the system assigns each resource metric a weight. If a container application heavily relies on a resource metric (e.g., IO intensive application relies on disk read/write rate more than network bandwidth), the system will assign a large value to this resource metric. Correspondingly, we change the random feature selection to weighted feature selection when choosing a feature of the data to divide the data set in the isolation forest algorithm. Thus, if a resource metric with a large weight is in an abnormal state, it will be more easily to be chosen as the feature to divide the data set. Therefore, the anomaly can be more accurately identified. When the anomaly value of a monitored data exceeds a predefined threshold, an anomaly is determined. Then, the system identifies the cause of the anomaly through analyzing the logs of the container. At the same time, the system can increase or decrease the monitoring period according to the degree of anomalies. Thus it can significantly reduce the alarm delay and monitor overheads.

The contributions of this paper are as follows:

- We design a docker container anomaly monitoring system that can monitor multidimensional resource metric, automatically adjust the monitoring period, and analyze the cause of the anomalies.
- We propose an optimized isolation forest algorithm that sets weights for different resource metrics and can locate the anomalous resource metric by taking into account the type of container application workload.
- We have implemented both the system and algorithm and evaluated them in both simulated and real commercial cloud (AWS) environments on a wide variety of anomaly cases in terms of detection accuracy, monitoring delay and log analysis.

2 Background and Related Work

In this Section, we first describe the background technologies on Docker and isolation forest. Then we elaborate the related work on the monitoring system and anomaly detection methods.

2.1 Docker Technology

Docker is a lightweight virtualization solution that is essentially a process on the host machine. Docker implements resource isolation through kernel-level namespaces. It allows process communications between hosts and containers without interfering with each other. Compared with virtual machines, Docker has the following advantages:

First, Docker has higher performance and efficiency than traditional virtualization methods. Unlike hardware-layer virtualization of virtual machines, Docker does not have hardware emulation, and implements virtualization at the operating system level [14].

Second, Docker has fewer layers of abstraction and does not require an additional Operating System (OS) and hypervisor support [15]. Thanks to this, Docker has better resource utilization. Typically, there can be thousands of Docker containers running on a single machine which can hold only a small number of virtual machines. Because of Docker's lightweight, the startup time only needs a few seconds, far faster compared with several minutes that a virtual machine needs.

Third, Docker can run on almost any platform, which makes Docker have better mobility and scalability [16]. In addition, it is easy to deploy and maintenance.

Due to the advantages of Docker over traditional virtual machines, more and more researchers begin to use Docker instead of virtual machines [16], [17], [18], [19]. For instance, Tihfon et al. [16] implemented the multi-task PaaS (Platform as a Service) cloud infrastructure with Docker, and they achieved rapid deployment of applications, application optimization and isolation. Nguyen et al. [18] implemented distributed Message Passing Interface (MPI) clustering for high-performance computing through Docker. Setting up MPI clusters was originally very time-consuming, but with Docker, they made this work relatively easy. Julian et al. [19] optimized the auto-scaling network cluster with Docker, and they believe that Docker containers can be used more widely in larger production environments.

2.2 Classic Isolation Forest Algorithm

Unlike other algorithms, the Isolation Forest algorithm (i.e., iForest [20]) does not need to define a mathematical model nor does it require training. It is somewhat similar to the dichotomy. The iForest consists of a number of isolation trees (i.e., iTree) where the leaf nodes are all single data. The sooner data is isolated, the more sparse it is in the data set, and therefore the more likely it is abnormal.

Assume that there are N data items in the data set. The steps of building an iTree are as follows:

First, we get n samples from the N data items as the training samples for this tree.

Second, we randomly select a feature, and randomly select a value p within the range of all values of this feature as the root node of the tree, then perform a binary division on the samples. The sample value that is smaller than p is divided into the left side of the root node, and the sample value that is greater than p is divided into the right side of the root node.

Third, we repeat the above process on the left and right data items until reach the termination condition. One is that the data itself cannot be divided (only one sample or all samples are the same), and the other is that the height of the tree reaches $log_2(n)$.

To make anomaly detection, we construct an iForest that consists of a number of iTrees. Assume the path length between each data x and the root node is $h(x)$, the average of all $h(x)$ is $E(h(x))$. $s(x,n)$ is the anomaly value of data x in the n samples of a data set. We compute it as follows:

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

$$c(n) = 2H(n-1) - (2(n-1)/n), H(k) = ln(k) + \xi. \tag{2}$$

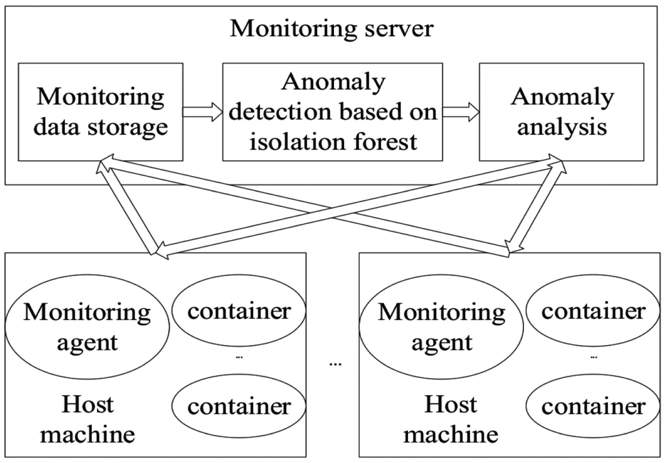


Fig. 1. System architecture.

The range of $s(x, n)$ is [0, 1]. The closer to 1, the higher the probability of an outlier is. The closer to 0, the higher the probability of x is normal. If most of the $s(x, n)$ are close to 0.5, the entire data set is considered to have no obvious outliers.

2.3 Monitoring System

Ganglia [6] is an open source cluster monitoring project initiated by UC Berkeley. Ganglia's main component includes gmond, gmetad, and a web front end. Gmond is installed on the monitored physical machine and is responsible for monitoring the collection of data. Gmetad is responsible for collecting data on the gmond and gmetad nodes. The web front end can show real-time data of the entire monitoring system. However, Ganglia can only provide monitoring and cannot analyze the cause of the anomalies.

Nagios [8] is a monitoring system that monitors system operating status and network information. It can monitor specified local or remote hosts and services, and provide exception notification functions. It can run on a Linux/Unix platform and also provides an optional browser-based web interface to allow system administrators to view network status, various system issues, and logs. Both Nagios and Ganglia need to set threshold, which is not suitable for monitoring a large number of containers in dynamically changing scenes.

Akshay et al. [12] proposed a simple container monitoring method that uses docker's own API to obtain resource and store it in the database. The method estimates the standard deviation of a resource monitoring parameter. The monitored data will be stored in the database only if the standard deviation exceeds a certain limit. This method has merit in data storage but lacks an alarm function.

cAdviosr [13] is a monitoring tool used by Google to provide a single-node multi-container resource monitoring function. As a running daemon, it collects, aggregates, processes, and exports information about running containers. It can obtain individual parameters and historical resource usage data for each container. Although cAdviosr is easy to set up and can generate charts, it can only monitor one Docker host and does not apply to a multi-node cluster environment. In addition, the chart data is just a one-minute sliding window. There is no data storage function, and no alarm function.

2.4 Anomaly Detection Method

The mathematical statistics-based method [9] builds some standard distribution models based on historical data, finds data points that deviate from distribution, and judges them as anomalies. However, most of the models are based on the assumption of a single variable. When the monitoring metric is multidimensional, it is difficult to accurately identify the anomaly. In addition, these models are calculated using the original data which contains noise data that has a significant impact on the building of the distribution model [21].

The information entropy based method [22] detects anomalies by comparing the entropies of the same cluster at different time. If there is a large fluctuation, it indicates the occurrence of anomalies. However, this method is only suitable for a stable operating environment. The dynamically changing container cluster will result in inaccurate detection results.

The idea of the distance-based method [23] is to calculate the distance between different data. When the distance between two data items is less than a neighbor distance D, they are regarded as "neighbors". If the number of neighbors of a data is less than the threshold p, then the data is judged to be an anomalous data. However, this method is not suitable for scenarios where the data distribution belongs to a multi-cluster structure [24]. Typically, multiple continuous anomalous resource metric data appear and cluster to be neighbors when an anomaly occurs. However, they cannot by identified by this method.

The most representative of the density-based methods is the Local Outlier Factor [10], which measures the degree of abnormality of each data instance based on the density-based local outlier factor. The larger the local outlier factor, the more likely it is abnormal. However, the local data density estimate can cause significant computational overhead when the sample data size is large [25]. Thus this is not suitable for a large number of containers.

3 System Design and Implementation

3.1 Architecture

The monitoring system architecture is shown in Fig. 1. It mainly consists of four components: *Monitoring agent*, *Monitoring data storage*, *Anomaly detection*, and *Anomaly analysis*.

There is only one *monitoring agent* on each host machine. It uses the non-invasive way to obtain the resource utilization rate of the container. The *monitoring data storage* module receives the monitoring data from each host. Only the monitoring data in the most recent period of time is stored, and the data is organized into a specified format and sent to the anomaly detection module. The *anomaly detection* module detects data received from the *monitoring data storage* module through a iForest-based abnormality evaluation method, and sends abnormal container information to the *anomaly analysis* module, which first obtains the log of the abnormal container from each host, then analyzes the log and locates the cause of the anomaly.

3.2 Monitoring Agent

The internal design of the *monitoring agent* is shown in Fig. 2. *Monitoring agent* collects the container data through the monitoring data collector. Then the monitoring agent

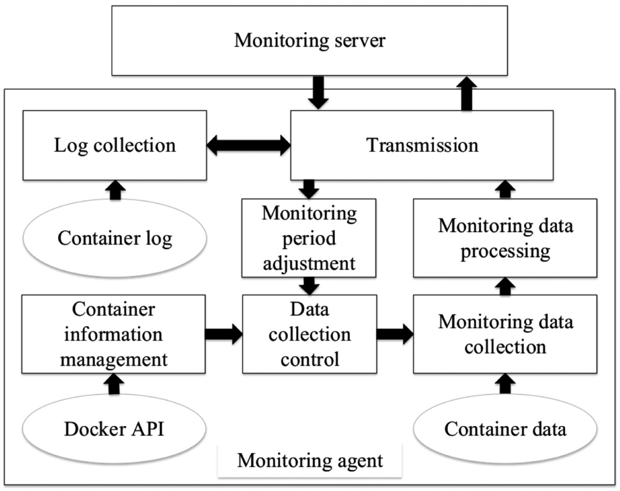


Fig. 2. Monitoring agent internal design.

communicates with the monitoring server, establishes a data transmission channel, and transmits the collected monitoring data through the channel. The monitoring cycle adjustment module will adjust the monitoring period according to the instructions from the monitoring server.

Monitoring Data Collection. This module is responsible for collecting monitoring data of all running containers on the host machine. The typical monitoring data includes the container's ID, time, CPU usage, memory usage, disk read/write speed, and network speed.

Monitoring Data Processing. This module receives the collected monitoring data from the monitoring data collection module. The module then performs two steps. The first step is to format the data and encapsulate the container's ID, time, and various resource usage into a format that the database can store directly. The second step is to check if there are identically mirrored containers running the same service and, if so, to summarize their monitoring data.

Container Information Management. This module mainly monitors the running status information of the container through the Docker API, including the startup of new containers, the close of old containers, their IDs, task information, and mirroring information. Then it passes these information to the data collection control module.

Monitoring Period Adjustment. The module maintains a data table, which contains the ID of each container on the host and its monitoring period. When receiving the monitoring period adjustment command sent by the server, the module changes the monitoring period and sends the changed results to the data collection control module.

Data Collection Control. This module is the control center of the monitoring agent and maintains a collection queue. It will calculate the next monitored container based on the last collection time and monitoring period of each container, and send this information to the monitoring data collection module. At the same time, the module also accepts the container start and stop information transmitted by the container information management module, thereby adding or deleting containers in the queue. The module can also adjust the monitoring sequence of the containers in the queue according to the monitoring period modification information transmitted by the monitoring period adjustment module. The monitoring period indicates the time interval to collect the container information. When a container is found to be likely to be abnormal, its monitoring period is reduced by half in order to identify the anomaly as soon as possible. In this case, the corresponding container information will be collected more frequently. Thus the container will be adjusted to a position in the front of the queue. In contrast, if a container recovers to normal, its monitoring period will double. The container will be adjusted to a position in the back of the queue.

Log Collection. Based on the log collection command from the monitoring server, the module collects logs for the specified container and passes the log to the transmission module in the specified format.

Transmission. It mainly has two functions: On one hand, it accepts various commands from the monitoring server and forwards the commands to the corresponding modules. On the other hand, it transfers the monitoring data to the monitoring server.

3.3 Monitoring Data Storage

The *monitoring data storage* module is responsible for storing the data collected by the *monitoring agent* and transmitting the data to the *anomaly detection* module in a specified format.

It uses InfluxDB [26] to store the collected container information. InfluxDB is an open source distributed timing, event and metrics database. It supports data transfer in the json format, thus facilitating data interaction with the monitoring agent and the anomaly detection module. A data table is created to store all the information of the containers. These information includes the container ID, the CPU usage, memory usage, disk read rate, disk write rate, network receive rate, network transmission rate of the container and data collection time. In order to save storage overhead, only the last hour of monitoring data is stored in the database.

The database also has a storage control table with three fields, the container ID, the number of rows in the data table, and the last modification time. There are three operations for the container information.

Creation and Insertion. After receiving the monitoring data sent by the monitoring agent, the container information is inserted into the data table. If the same container ID is not found in the data table, it indicates that the monitoring data is from a newly opened container. The database will create a new row in the storage control table to add the information of the new container. If the same ID is found, the *number of rows* and the modification time of the corresponding container in the storage control table is modified.

Deletion. The storage control table is scanned for every ten minutes. When it is found that the information of a container has not been updated for more than ten minutes, it is judged that the container has been closed, and the database deletes the corresponding container information in both the data table and the storage control table.

Sending Data to the Anomaly Detection Module. Because in the anomaly detection module, a certain amount of data is needed to build an isolation forest. When the value of *number of rows* in the storage control table for a container reaches 100, 100 rows of data in the data table for this container are sent to the anomaly detection module in json format.

TABLE 1
Dirty Data Type

Category	Dirty data manifestations
Missing value	One of the data is null
Repeat value	Redundant data appears
Maximum or minimum	Suddenly the data is too big or too small

3.4 Anomaly Detection

3.4.1 Data Cleaning

Due to the large amount of container data to be collected, there may be data loss, duplication, or changes in transit and storage. Therefore, before constructing an isolation forest, it is necessary to first clean the data and remove the dirty data inside. Common dirty data types are shown in Table 1:

The first is to delete the redundant data in the data set. Redundant data can affect the structure of isolation forests and reduce the accuracy of anomaly detection. When multiple identical records appear, the extra data must be deleted.

In addition, the integrity of the data set must be preserved. The absence of data often occurs in datasets and must therefore be handled appropriately, or else it will affect the structure and anomaly detection accuracy of isolation forests. Severe missing cases are defined as: a) Missing more than 20 percent of monitoring points over a period of time. b) Missing consecutive 5 or more monitoring points.

If there is a serious loss of data in the data set, the data in that period is excluded from the detection range.

3.4.2 Optimization of Isolation Forest Algorithm

Introduction and Calculation of Resource Weight. The idea of the classic iForest algorithm has been very concise and efficient, and can be directly applied to many application scenarios. However, there are still some problems when it is applied to the container environment. In container monitoring, there are four most commonly used monitoring indicators: CPU usage, memory usage, disk read and write rates, and network speed. When the iForest algorithm is applied to the container monitoring, these four indicators become the features used to divide the data set. However, in the classic iForest algorithm, the probability of being selected is the same for all features in the random case. In the container environment, the container applications that are CPU-intensive are more dependent and sensitive to CPU resources, and the container applications that are IO-intensive are more dependent and more sensitive to IO. If containers that rely on different kinds of resources are biased to use the same standard for monitoring, it is inevitable that anomaly detection will not be inaccurate.

Therefore, this paper designs an optimization method. The basic principle of this optimization is to set a weight value for each of the four resource indicators, and then to change the random selection to weighted randomness when selecting features in the construction of isolation trees. In this way, resource indicators with high weights are more likely to be selected for data classification than other indicators. Therefore, the anomalies in containers that are more dependent and more sensitive to such resources are more likely to be found.

Here, a self-learning method for resource bias optimization is proposed. During the normal use of a container, the container's bias parameters M for each resource is calculated as formula (3):

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

W_0 is the initial weight value of the resource metric, and its value is 1. ϵ is the resource threshold. N_i is the usage rate of the resource at the time i. p is the number of times to measure the resource usage. If $x > 0$, then $f(x) = 1$, otherwise $f(x) = 0$. If the value of the resource metric is always 0, the container does not use the resource. So we set its weight to 0. The larger the parameter M, the more the container is biased toward the resource.

The bias parameter M is used as the weight value for each resource metric. First of all, by default, all resource indicators have a weight value of 1. Then we determine the period under which the weight value is modified. We specify every 10 minutes as a period. The bias parameter M is calculated by the data usage rate during this period, and then the weight value is replaced by M. Finally, a weighted random algorithm is used to select the eigenvalues. The pseudocode of the algorithm is shown in Algorithm 1.

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: end for
6: return $i$
```

M_1, M_2, M_3, and M_4 are the four resource weight values. M_{all} is the sum of all weight values. R is a random data in the range of 0 to M_{all}, and the last returned i is an index number of the resource selected as a feature to divide the data set.

Anomaly Resource Metric Judgement. The iForest algorithm can calculate the anomaly value of the multidimensional resource metrics, but cannot determine which metric causes the anomaly. For example, there are two kinds of exception cases, one is that the CPU usage is abnormally increased, and the other is that the memory usage is abnormally increased. The anomaly value is similar in both cases using the iForest algorithm. It is impossible to distinguish which kind of anomaly in resource usage that has caused this. In order to solve this problem, this paper proposes a method to judge the anomaly metric.

1) When constructing an isolation tree, if a leaf node is generated when a division is performed, the feature selected by the division is called an isolation feature of the data on the leaf node, indicating that this data is isolated by this feature in the last division.
2) Set an isolation feature group for each data, such as $S(S_1, S_2, ..., S_n)$. S_i represents the number of times

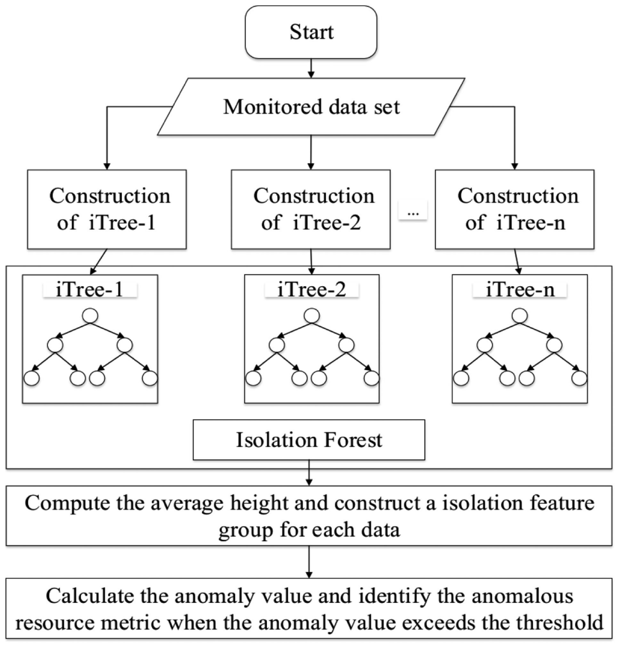

Start
Monitored data set
Construction of iTree-1
Construction of iTree-2
...
Construction of iTree-n
iTree-1
iTree-2
iTree-n
Isolation Forest
Compute the average height and construct a isolation feature group for each data
Calculate the anomaly value and identify the anomalous resource metric when the anomaly value exceeds the threshold

Fig. 3. Isolation forest construction process.

the metric feature numbered i is used as an isolation feature of the data in the isolation forest.

When we repeatedly construct isolation trees and make a summarize of the isolation feature group for each data, the resource metric with a higher value in the isolation feature group is more likely to be anomalous than the resource metric with a lower value. Thus it can be judged which resource metric mainly caused the increase in the anomaly value of the monitoring data.

The method is based on a premise: if a feature value of a data has a large difference from the value of this feature of other data, then when dividing by this feature, this data is more likely to be isolated separately. Therefore, it can be inferred that the isolation feature of a data is also the feature that is most likely to have the biggest anomalous value.

When it is determined that the container is abnormal, the isolation feature group of the anomalous monitoring data and the isolation feature group of the normal monitoring data are compared. We calculate the ratio of the corresponding values of the metrics in the isolation feature groups. The higher the ratio, the higher the degree of anomaly of the metric.

Construction of iTree and iForest. Isolation forest consists of many iTrees. iTree is a kind of random binary tree. Each node has either two child nodes or is a leaf node itself. Leaf nodes are isolated data. This article uses the container's CPU usage, memory usage, IO read/write rates, and network rate as four features for constructing an isolation tree.

iTree construction steps are as follows:

1) Calculate the bias of each resource of the current container based on the monitoring data, and modify the corresponding feature weight;
2) Select a feature F among the four container resource features. (i.e., CPU usage rate, Memory usage rate, IO rate and write rate, Network rate) according to Algorithm 1;
3) Randomly select a value n from the range of the value of feature F;
4) According to the feature F, the data set is divided. The data with the value of feature F less than n are divided into the left branch, and the data with the value of feature F greater than or equal to n are divided into the right branch.
5) Repeat steps 2) through 4) recursively to construct the left and right branches of the iTree until the following conditions are met:
   a) There is only one data in the data set to be split;
   b) The height of the tree reaches a predefined height

As shown in Fig. 3, the construction of the isolation forest is somewhat similar to the random forest. Each part of the data set is randomly sampled to construct each tree. Then we calculate the average height of each data in all the itrees and compute the anomaly value of the data according to formulas (1) and (2). We can further compute the number of times that each resource metric is used as the isolation feature and identify the anomalous resource metric.

3.4.3 Monitoring Period Adjustment

In order to improve the timeliness of monitoring, the monitoring period can be reduced to collect more monitoring data to detect changes in the monitoring data anomaly value earlier in the case of possible anomalies. An anomaly sensitivity threshold f is set to determine whether an anomaly is likely to occur. The value of f is related to the anomaly detection threshold d and can be expressed as:

$$f = \frac{d + p}{2}. \tag{4}$$

p is the normal anomaly value originally set for the isolation forest and is set to 0.5 by default. When the average value of the anomaly value of the data in a period is between f and d, although the criterion for judging the anomaly is not reached at this time, the high anomaly value indicates that the container may be abnormal. At this time, the container is set as an intensive monitoring object, and the monitoring server sends a message such as {"container_id": 100 ; "type": intensive} to the monitoring agent. The container_id is the ID of the container, and there are two types: *intensive* and *extensive*. When the type is *intensive*, the corresponding monitoring period is set to half of the initial monitoring period. When the average value of the anomaly value of the data is lower than f, the command of type *extensive* is sent to the *monitoring agent* to adjust the monitoring period to the initial monitoring period.

3.5 Anomaly Analysis

The *anomaly analysis* module mainly analyzes the log of the abnormal container identified by the *anomaly detection* module, and finds why the anomaly is caused. The source data for the anomaly analysis are the log collected by the log collection module in the monitoring agent. The *anomaly analysis* module mainly contains the following two parts.

3.5.1 Log Preprocessing

Before the log analysis, the first step is to perform log preprocessing. We extract only useful log events to reduce storage overhead and analysis overhead.

TABLE 2
Configuration Information of the Experiment

Machine	Hardware Configuration	Software Configuration
1	Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz, 16 Cores, 32G RAM	Ubuntu 16.04 Docker 18.03.1-ce InfluxDB 0.13.0 MySQL 5.7 Logstash 6.2.4
2	Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz, 16 Cores, 32 G RAM	Ubuntu 16.04 Docker 18.03.1-ce Memcached v1.5.7 CloudSuite v3.0 Logstash 6.2.4

The system log of the container is directly saved in json format, which will generate a large number of escape sequences such as $u0008$. This greatly increases the amount of logs. Therefore, the corresponding filtering process should be performed on such escape sequences. There are also many events in the application log that are not related to exception analysis. For example, the web application logs records the access logs (such as access on jpg files) that have no effect on the anomaly analysis. And this part needs to be filtered. The specific operation of log filtering is to configure regular expression matching in the filter plugin of the logstash [27] configuration file, and then use the drop operation to delete the matching corresponding log content. Then the filtered log data will be stored into database.

3.5.2 Log Analysis

The main function of the log analysis module is to mine the frequent itemsets of the pre-processed log events, compare them with the rule database, find out the log events that caused the exceptions, and update the rule database. The rule database includes two types: the normal rule database and the exception rule database. The rules in the normal rule database represent the frequent itemsets generated when the container is running normally. The rules in the exception rule database are divided into two types. One is an empirical exception rule, which is an exception filtering condition added by experience, such as a log level of ERROR, or a regular expression that can find a typical abnormal log event by matching. The other is a historical anomaly rule, which is obtained by filtering the frequent itemsets of the log that were previously analyzed and caused by the administrator.

The basic flow of log analysis is as follows:

First, we match the log stored in the database with the empirical exception rules in the exception rule database. If the match is successful, the log event alarm is output. Otherwise, the Apriori algorithm [28] is used to mine the frequent itemsets in the log transaction.

Second, we match the frequently mined itemsets with the normal rules and the historical exception rules. If it matches the normal rules, it is filtered out. If it matches the historical exception rules, the log event alarm corresponding to the frequent itemsets is output.

Third, if none of the matches is successful, the administrator selects the frequent itemsets and adds them to the normal rule database and the exception rule database.

TABLE 3
Species of Anomalies

Classification of anomalies	Illustration
Anomalies about CPU	Endless loop, spin lock
Anomalies about memory	Memory leak, memory overflow
Anomalies about disk	Improper disk scheduling, log explosion
Anomalies about net	Network attack, network congestion

4 Experimental Evaluation

4.1 Experimental Environment

We do experiments in both simulated and real cloud environments. For the simulated cloud environment, we deploy monitoring server in one machine, and monitoring agent and Docker container in the other machine. The configuration information is shown in Table 2. For the real cloud environment, we adopt the Amazon EC2 service [29]. We use two types of configurations. One type is called t3. medium with 2 CPU cores and 4 GB RAM. Another type is for free and it is called t3.small with limited use of 1 CPU core and 2 GB RAM. Both of the platforms run Ubuntu 16.04 and Docker 18.03.1-ce. All the monitoring components run in the cloud platform.

We demonstrate the monitoring system with two representative benchmarks in cloud environment. One of them is Memcached, and the other one is Web Search in CloudSuite. Memcached is an open source, high-performance, distributed memory object caching system and intended for use in speeding up dynamic web applications by alleviating database load [30]. CloudSuite is a benchmark suite for cloud services and consists of eight applications that have been selected based on their popularity in today's data centers [31]. The Web Search benchmark is one of them and relies on the Apache Solr search engine framework. It contains a 12 GB index which was generated by crawling a set of websites with Apache Nutch. For Memcached, we use Mutilate [32] as a workload generator, and for Web Search, we use Faban client provided by CloudSuite as a workload generator.

Since there is no benchmark for container anomaly injection, we divided anomaly into four common categories that involve different resource metrics. They are shown and illustrated in Table 3.

Similar to the previous work [33], we use the following four cases to simulate the anomalies.

Endless Loop in CPU. We inject this fault in the application by inserting additional code to call stress tool [34], which can simulate an endless loop in the CPU and take up CPU utilization of 100 percent .

Memory Leak. The injected code allocates heap memory without releasing objects, which can gradually take up 100 percent of memory utilization.

Disk I/O Fault. We use FIO [35] to inject extra operations of reading and writing disk and simulate disk I/O fault.

Network Congestion. We simulate network congestion by using wondershaper [36] to limit the bandwidth of the specified network interface.

4.2 The Result Comparison of Anomaly Detection

We use detection rate and false alarm rate to evaluate the result of anomaly detection.

TABLE 4
The Result Comparison of Anomaly Detection on Memcached and Web Search

	Anomalies	Original iForest		Optimized iForest		LOF	
		Detection rate	False alarm rate	Detection rate	False alarm rate	Detection rate	False alarm rate
Memcached	Endless loop in CPU	38%	0%	100%	2%	100%	5.8%
	Memory leak	30%	0%	98%	2%	85%	2.3%
	Disk I/O fault	46%	4.2%	76%	5%	54%	6.9%
	Network congestion	94%	2.1%	100%	2%	100%	2%
Web Search	Endless loop in CPU	48%	7.7%	100%	5.7%	100%	12.3%
	Memory leak	40%	4.8%	100%	7.4%	96%	14.3%
	Disk I/O fault	42%	5.7%	72%	12.2%	58%	23.7%
	Network congestion	74%	5.1%	84%	6.7%	80%	14.9%

$$P_{detection\ rate} = \frac{TP}{TP + FN} \times 100\% \tag{5}$$

$$P_{false\ alarm\ rate} = \frac{FP}{TP + FP} \times 100\%. \tag{6}$$

TP (true positive) indicates the number of anomalies which are classified correctly. FN (false negative) represents the number of anomalies which are not identified. FP (false positive) summarizes the normal behaviors that have been judged as anomalies.

In order to test the detection result of the proposed method, two other detection methods are used as comparisons. One is original iForest-based anomaly detection method, and the other is based on local anomaly factor algorithm (i.e., LOF [10]) which is the most representative density-based anomaly detection method. 200 tests were performed and each of the four typical anomalies mentioned above is injected 50 times.

Tables 4 summarize the result of anomaly detection for different methods on Memcached and Web search respectively. The results show that the optimized iForest has a lower false alarm rate on Memcached compared to Web Search. This is because the Memcached container's resource metric under the normal load is very stable. When an anomaly occurs, the anomaly value of the monitoring data changes greatly, so it has a high detection accuracy. The fluctuation in the resource metric of Web Server under the normal load is not small, and sometimes continuous fluctuations will cause the anomaly value to rise beyond the anomaly detection threshold, resulting in false alarms.

The optimized iForest has a significant improvement on detection rate compared to the original iForest. This is because the anomalous resource metric in optimized iForest is assigned a large weight and thus more easily to be chosen as the isolation feature to divide the data set. The average height of the data divided using the isolation feature in iForest is thereby very small, resulting in a big anomaly value. Thus the detection rate of the optimized iForest is high.

The optimized iForest has a comparable or better performance than LOF. For instance, under the injection of Disk I/O fault, the detection rate of LOF is significantly lower than that of optimized iForest. It is because anomalous disk read or write rate is not much different from normal disk read or write rate which has a small fluctuation. So the local density of monitored data has only a little change and thus detection rate of LOF is low. Besides, LOF has a higher false alarm rate compared to optimized iForest, especially on Web Search. It indicates LOF is more susceptible to fluctuant resource metrics at normal runtime.

The above experiments assume that the injected malicious programs consume 100 percent of CPU by endless loops. However, in practical, the malicious user who tries to compromise the performance of whole system can use malicious programs that not only take 100 percent of CPU but, for example, 60 percent of CPU for a long time. Table 5 shows the performance results in this case for two types of cloud environments. Cloud1 and Cloud2 represent the different cloud platforms with multiple cores and single core respectively. For Cloud1, we use the siege tool [37] to simulate the web attack that consumes 60–80 percent CPU resource. For Cloud2, we find that the siege tool cannot increase the CPU utilization by 60 percent. Instead, we execute a program with 500 thousand times of loops. For each loop, the program sleeps for 0.1 milliseconds. The optimized iForest performs the best on detection rate in both of the two cloud environments. Though the original iForest has no false alarms, it cannot detect the anomaly caused by the malicious program in most of the time. Comparatively, the optimized iForest has an acceptable small false alarm rate. The false alarm rate in Cloud2 is larger than in Cloud1 for the optimized iForest. The possible reason is that there exists more fluctuations in the resource metrics in Cloud2.

Overall, optimized iForest has better anomaly detection results compared to other two methods.

TABLE 5
The Result Comparison of Anomaly Detection when the Malicious Program Consumes CPU Utilization that Exceeds 60 percent

Platforms	Categories	Original iForest	Optimized iForest	LOF
Cloud1	Detection rate	24%	100%	100%
	False alarm rate	0%	1.96%	4.76%
Cloud2	Detection rate	16%	100%	79%
	False alarm rate	0%	3.84%	11.23%

Cloud1: Amazon EC2, t3.medium, 2 CPU, 4 GB RAM; Cloud2: Amazon EC2, t3.small, 1 CPU, 2 GB RAM.

4.3 A Case for Anomaly Detection

Here is an example showing how to detect anomaly in Memcached container. During the period of running in Memcached

TABLE 6
The Weights of Resource Metrics

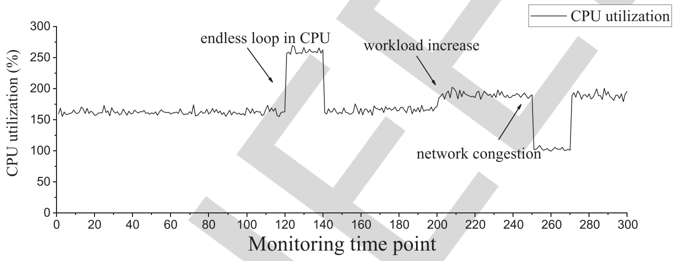
Resource metric	Weight
CPU utilization	2
Memory utilization	1
Disk read rate	0
Disk write rate	0
Network receive rate	2
Network transmit rate	1

container, three events are inserted. Two of them are anomalies, which are the endless loop of CPU and network congestion. The other event is the workload increase. The calculated weights of resource metrics are shown in Table 6.

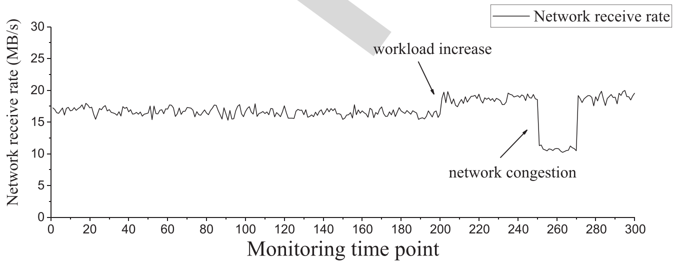
Fig. 4 illustrates the CPU utilization and network receive rate monitored at Memcached containers runtime. Note that in a system with multiple cores where the container applications are running, the CPU utilization can exceed 100 percent. Actually, in a docker system with n cores, the total system CPU utilization can be 0–n*100% [38], [39].

Fig. 5 illustrates the variation of anomaly indexes calculated according to monitor metrics. It shows that when an endless loop in the CPU is injected, the anomaly indexes increase significantly. The average value of the anomaly indexes between the monitoring time point at 121 and 130 is 0.585, which exceeds the detection threshold in red line. Thus the container is identified as anomalous. When network congestion is injected, the anomaly indexes increase significantly. The average value of the anomaly indexes between the 251th and 260th monitoring points is 0.582, which exceeds the detection threshold. And the container is identified as anomalous. However, the workload increase does not make the anomaly index increase, and thus it is not identified as an anomaly.

The anomalous resource metric needs to be located after detecting container anomaly. We propose a method that calculates the ratio of isolation features in the anomalous phase to isolation features in the normal phase. Table 7 shows the ratio of isolation features when endless loop in CPU and network congestion are injected. It can be seen that the ratios of isolation features for anomalous resource metrics are higher than others. So this method can accurately locate the anomalous resource metric.



(a) The CPU utilization of Memcached container at runtime



(b) Network receive rate of Memcached container at runtime

Fig. 4. Resource metrics monitored at Memcached containers runtime. Note that in a docker system with n cores, the total system CPU utilization can be 0–n*100% [38], [39]. The value of n is 16 in this experiment.

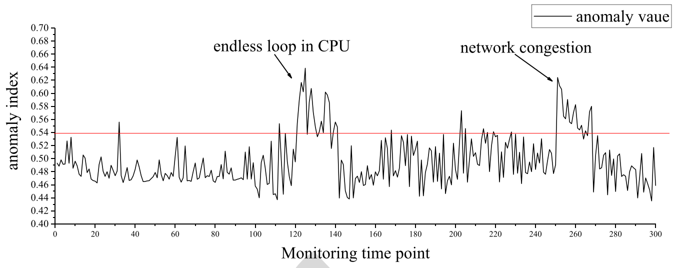


Fig. 5. Anomaly values of Memcached container at runtime. The red line shows the detection threshold.

4.4 Detection Threshold d

The detection rate and false alarm rate are closely related to the detection threshold d. In order to find the optimal value, 200 tests were performed, including the four typical anomalies mentioned above and each of them was performed 50 times. Different detection thresholds were used for detection. The results are shown in Fig. 6.

Both the detection rate and false alarm rate decrease rapidly with the increase in d. We need to choose the value of d with a high anomaly detection rate and a low false alarm rate. According to the Fig. 6, the optimal value of d is 0.54.

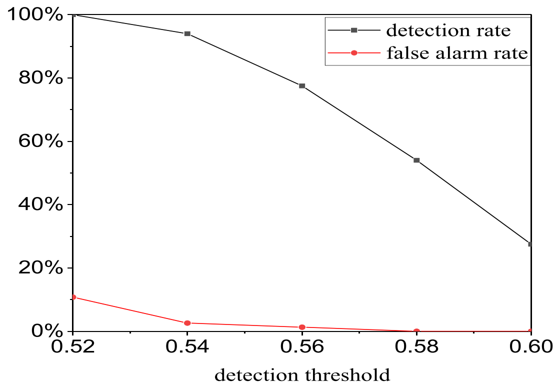
4.5 The Number of iTrees

The number of iTrees is an important parameter in the optimized iForest. In order to find its optimal value, we measure the detection rate and the false alarm rate and the computation time under different numbers of iTrees. The detection threshold is set as 0.54. The results are shown in Fig. 7.

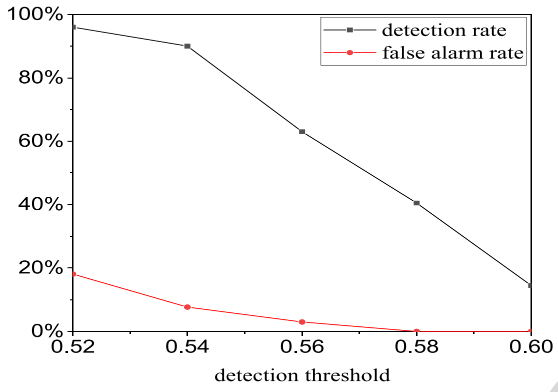
It can be seen that the detection rate increases and the false alarm rate decreases as the number of iTrees increases. But the computation time still increases proportionally. Increasing the number of iTrees does not improve anomaly detection effect after the number of iTrees is bigger than 100. So the optimal value of the number of iTrees is 100.

TABLE 7
Ratio of Isolation Features when Endless Loop in CPU and Network Congestion are Injected

	Resource metric	Ratio of isolation features
endless loop in CPU	CPU utilization	1.23
	Memory utilization	0.81
	Network receive rate	0.89
	Network transmit rate	0.83
network congestion	CPU utilization	1.13
	Memory utilization	0.75
	Network receive rate	1.16
	Network transmit rate	0.78



(a) Memcached

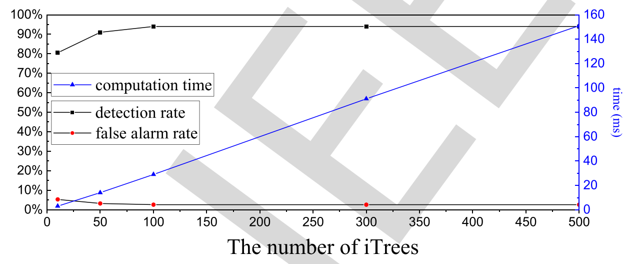


(b) Web Search

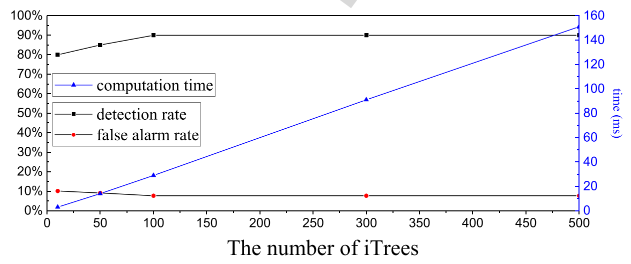
Fig. 6. Anomaly detection effect diagram in the case of different detection threshold d.

4.6 Monitoring Delay

The interval between when the anomaly is injected and when the anomaly is found is defined as the monitoring delay. Two sets of anomaly detection tests based on the optimized iForest are performed. One of tests uses the fixed monitoring period of 4 seconds, i.e., we get a group of container data every 4 seconds. The other test adopts the method of dynamically adjusting the monitoring period. The initial monitoring period is also 4 seconds. We inject four typical anomalies mentioned above for each test. The comparison results are shown in the Fig. 8.



(a) Memcached



(b) Web Search

Fig. 7. Anomaly detection results with different numbers of iTrees.

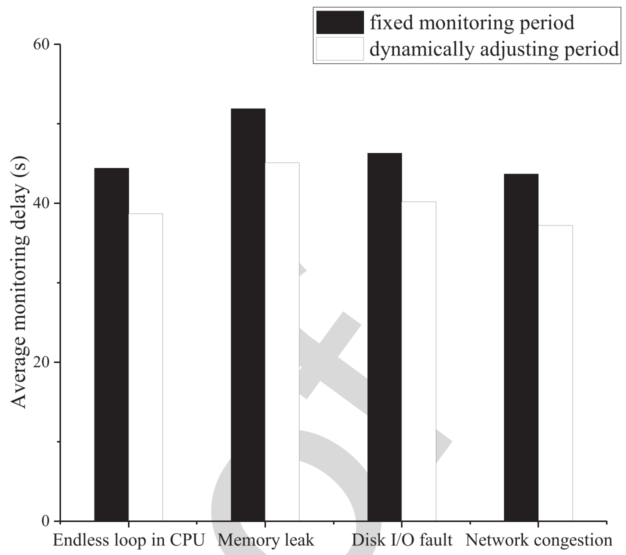


Fig. 8. Average monitoring delay comparison.

The monitoring delay of dynamically adjusting period is significantly lower than the monitoring delay of fixed monitoring period. When an anomaly is identified, the monitoring period reduces by half. More monitoring data is collected in a unit of time, making the anomaly detected earlier. When the container recovers to the normal status, the monitoring period is adjusted to the initial value. The dynamically adjusting period reduces monitoring delay by an average of 13.5 percent.

The average monitoring delays are between 40 and 55 seconds while the setting of monitoring period is fixed 4 seconds. The reason is as follows. The optimized iForest algorithm initially gets 100 groups of data to build an iForest. It has a window size of 100 and a sliding distance of 10. Whenever it gets 10 new groups of data, it uses previously 90 groups of data and this 10 new groups of data to build a new iForest. If the average anomaly value of this 10 groups of data exceeds the detection threshold, an anomaly can be identified. As it takes 4 seconds to get a group of data, it needs a total of 40 seconds to get this 10 groups of container data. Thus when the anomaly of these data is identified, the monitoring delay is at least 40 seconds. Comparatively, when the monitoring period can be dynamically adjusted, the monitoring period can be below 4 seconds. Thus the monitoring delay can be lower than 40 seconds sometimes.

4.7 Cases for Log Analysis

Here are two examples showing how to analyze containers logs. In order to locate the cause of anomaly by analyzing logs, two anomalies which leave traces in the logs are injected. One is reading and writing disk constantly using postmark to simulate the disk attack. The other is to send a large number of GET requests to the webpage to simulate the network attack.

Disk Attack. When postmark is running constantly, the disk read-write rate increases abnormally, and the container is identified as anomalous. Then the anomaly analysis module collects anomalous containers system log. After pre-processing, the size of log diminishes from 476 KB to 143 KB. Then the log is stored in the database.

The result of association rule analysis is:

Creating files...Done stdout —(frequency)——>>146

Data: stdout —(frequency)——>>147

Deleting files...Done stdout —(frequency)——>>146

It indicates there are 146 logging events including Creating files and 147 logging events including Data and 147 logging events including Deleting files. It can be inferred the container creates and deletes files frequently in anomalous phase.

Network Attack. In this experiment, a nginx container starts with a website running in it. To simulate network attack, an anomaly injection program is performed to send a mass of GET requests to the website. Then the network send/receive rates increase abnormally, and the container is identified as anomalous. The anomaly analysis module collects anomalous containers application log. After pre-processing, the number of logging events diminishes from 1434 to 723.

The result of association rule analysis is:

/ 192.168.220.1 200 GET —(frequency)——>>137

It indicates the cause of anomaly is that a host whose IP is 192.168.220.1 sends 137 GET requests to the website.

5 Conclusions

This paper proposes an online container anomaly detection system by monitoring and analyzing multidimensional resource metrics of the containers based on optimized isolation forest algorithm. To improve the detection accuracy, it assigns each resource metric a weight and changes the random feature selection in the isolation forest algorithm to the weighted feature selection according to the resource bias of the container application. The monitoring period can be dynamically adjusted according to the degree of abnormality to reduce the monitoring delay. In addition, it collects and analyzes log for the cause of the anomalies. The experimental results on both simulated and real cloud platforms show that the method can accurately detect anomalies in the container with small performance overheads.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Science Foundation of China under Grant No. U1705261, 61972449, and 61821003, CCF-NSFOCUS Kun Peng research fund, Wuhan Application Basic Research Program under Grant No. 2017010201010104, and Hubei Natural Science and Technology Foundation under Grant No. 2017CFB304, and the Fundamental Research Funds for the Central Universities under Grant No. 2019kfyXKJC021.

References

[1] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, 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 *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 265–278.

[2] “Amazons container strategy examined.” [Online]. Available: https://www.informationweek.com/cloud/infrastructure-as-a-service/amazons-container-strategy-examined/a/d-id/1317515

[3] “IBM containers on bluemix.” [Online]. Available: https://www.ibm.com/blogs/bluemix/2015/06/ibm-containers-on-bluemix/

[4] “Munz docker occs.” [Online]. Available:http://www.oracle.com/technetwork/articles/cloudcomp/munz-docker-occs-3585210.html

[5] “Amazons cloud service partial outage affects certain websites.” [Online]. Available: http://www.dailymail.co.uk/sciencetech/article-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 Comput.*, vol. 30, no. 7, pp. 817–840, 2004.

[7] “Introduction to Zabbix.” [Online]. Available: http://tim.kehres.com/

[8] “Nagios: The industry standard in it infrastructure monitoring.” [Online]. Available:https://hops://www.n-agios.org/

[9] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM Comput. Surv.*, vol. 41, no. 3, pp. 1–58, 2009.

[10] M. M. Breunig, H. P. Kriegel, and R. T. Ng, “LOF: Identifying density-based local outliers,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 93–104.

[11] H. P. Kriegel, M. S. Hubert, and A. Zimek, “Angle-based outlier detection in high-dimensional data,” in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2008, pp. 444–452.

[12] S. A. K, J. A. K, and K. A, “Resource monitoring of docker containers,” *Int. J. Adv. Eng. Res. Develop.*, vol. 3, no. 2, pp. 146–149, 2016.

[13] “Google.cAdvisor.” [Online]. Available: https://github.com/google/cadvisor

[14] N. Naik, “Building a virtual system of systems using docker swarm in multiple clouds,” in *Proc. IEEE Int. Symp. Syst. Eng.*, 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,” in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, 2015, pp. 490–491.

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

[17] R. Liu, R. Liu, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Slacker: Fast distribution with lazy docker containers,” in *Proc. Usenix Conf. File Storage Technol.*, 2016, pp. 181–195.

[18] N. Nguyen and D. Bein, “Distributed MPI cluster with docker swarm mode,” in *Proc. Comput. Commun. Workshop Conf.*, 2017, pp. 1–7.

[19] S. Julian, M. Shuey, and S. Cook, “Containers in research: Initial experiences with lightweight infrastructure,” in *Proc. Xsede16 Conf. Diversity Big Data Sci. Scale*, 2016, Art. no. 25.

[20] F. T. Liu, M. T. Kai, and Z. H. Zhou, “Isolation forest,” in *Proc. 8th IEEE Int. Conf. Data Mining*, 2009, pp. 413–422.

[21] N. L. D. Khoa and S. Chawla, *Robust Outlier Detection Using Commute Time and Eigenspace Embedding*. Berlin, Germany: Springer, 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 Trans. Dependable Secure Comput.*, vol. 8, no. 4, pp. 510–522, Jul./Aug. 2011.

[23] S. Ramaswamy, R. Rastogi, and K. Shim, “Efficient algorithms for mining outliers from large data sets,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 427–438.

[24] F. Angiulli, S. Basta, and C. Pizzuti, “Distance-based detection and prediction of outliers,” *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 2, pp. 145–160, Feb. 2006.

[25] D. Pokrajac, A. Lazarevic, and L. J. Latecki, “Incremental local outlier detection for data streams,” in *Proc. IEEE Symp. Comput. Intell. Data Mining*, 2007, pp. 504–515.

[26] “Influxdb - an open source distributed time series database.” [Online]. Available: https://www.infoq.com/fr/presentations/influx-db/

[27] S. Sanjappa and M. Ahmed, “Analysis of logs by using logstash,” in *Proc. 5th Int. Conf. Frontiers Intell. Comput.: Theory Appl.*, 2017, pp. 579–585.

[28] “Apriori algorithm.” [Online]. Available:https://en.wikipedia.org/wiki/Apriori_algorithm

[29] “Amazon elastic compute cloud (amazon ec2).” [Online]. Available:http://aws.amazon.com/ec2

[30] “What is Memcached.” [Online]. Available:http://memcached.org/

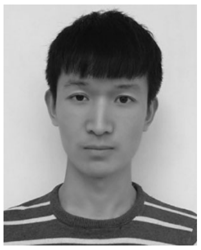
[31] “A benchmark suite for cloud services.” [Online]. Available: http://cloudsuite.ch/

[32] "Leverich.Mutilate." [Online]. Available:https://github.com/leverich/mutilate
[33] T. Wang, W. Zhang, J. Wei, and H. Zhong, "Workload-aware online anomaly detection in enterprise applications with local outlier factor," in *Proc. IEEE 36th Annu. Comput. Softw. Appl. Conf.*, 2012, pp. 25–34.
[34] "stress." [Online]. Available:http://people.seas.harvard.edu/ apw/stress/
[35] "FIO." [Online]. Available:http://freshmeat.sourceforge.net/projects/fio/
[36] "wondershaper." [Online]. Available:http://lartc.org/wondershaper
[37] "Siege home." [Online]. Available:https://www.joedog.org/siege-home/
[38] "The moby project." [Online]. Available:https://github.com/moby
[39] "Docker community forums." [Online]. Available:https://forums.docker.com/t/docker-stats-questions/811

Zhuping Zou received the BE degree in computer science from Central South University of Forestry and Technology, China, in 2017 and the master's degree from Huazhong University of Science and Technology (HUST), in 2019. His research interests include docker container and virtualization.

Yulai Xie received the BE and PhD degrees in computer science from Huazhong University of Science and Technology (HUST), China, in 2007 and 2013, respectively. He was a visiting scholar with the University of California, Santa Cruz, in 2010 and a visiting scholar with the Chinese University of Hong Kong, in 2015. He is now an associate professor with HUST, China. His research interests mainly include cloud storage and virtualization, digital provenance, intrusion detection, machine learning, and computer architecture. He is a member of the IEEE.

Kai Huang received the master degree from Huazhong University of Science and Technology, in 2018. His research interests include docker container and virtualization.



Gongming Xu received the BE degree in computer science from Wuhan Institute of Technology, China, in 2018. He is currently working toward the master's degree at Huazhong University of Science and Technology (HUST).

Dan Feng received the BE, ME, and PhD degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 1991, 1994, and 1997, respectively. She is a professor and director of Data Storage System Division, Wuhan National Lab for Optoelectronics. She is also dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, parallel file systems, disk array and solid state disk. She has more than 100 publications in journals and international conferences, including FAST, USENIX ATC, ICDCS, HPDC, SC, ICS and IPDPS. She is a member of the IEEE and a member of ACM.

Darrell Long received the BS degree in computer science from San Diego State University, and the MS and PhD degree from the University of California, San Diego. He is a distinguished professor of computer engineering with the University of California, Santa Cruz. He holds the Kumar Malavalli endowed chair of Storage Systems Research and is director of the Storage Systems Research Center. His current research interests include storage systems area include high performance storage systems, archival storage systems and energy-efficient storage systems. His research also includes computer system reliability, video-on-demand, applied machine learning, mobile computing and cyber security. He is fellow of the IEEE and fellow of the American Association for the Advancement of Science (AAAS).