This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2016.2520883, IEEE Transactions on Emerging Topics in Computing

Non-intrusive Anomaly Detection with Streaming Performance Metrics and Logs for DevOps in Public Clouds: A Case Study in AWS

Daniel Sun, Min Fu, Liming Zhu, Guoqiang Li and Qinghua Lu *Member, IEEE*

Abstract—*Public clouds* are a style of computing platforms where scalable and elastic IT-enabled capabilities are provided as a service to external customers using Internet technologies. Using public cloud services can reduce costs and increase choices of technologies, but it also implies limited system information for users. Thus, anomaly detection at user end has to be non-intrusive and hence difficult, particularly during DevOps operations because the impacts from both anomalies and these operations are often indistinguishable and hence it is hard to detect the anomalies. In this paper, our work is specific to a successful public cloud, Amazon Web Service (AWS), and a representative DevOps operation, rolling upgrade, on which we report our anomaly detection that can effectively detect anomalies. Our anomaly detection requires only metrics data and logs supplied by most public clouds officially. We use Support Vector Machine (SVM) to train multiple classifiers from monitored data for different system environments, on which the log information can indicate the best suitable classifier. Moreover, our detection aims at finding anomalies over every time interval, called window, such that the features include not only some indicative performance metrics but also the entropy and the moving average of metrics data in each window. Our experimental evaluation systematically demonstrates the effectiveness of our approach.

Index Terms—Public clouds, Non-intrusive anomaly detection, Performance metrics, Logs, Machine Learning.

1 Introduction

Anomaly detection refers to the problem of finding patterns in data that do not conform to expected behaviour [1], and has been an indispensable element of large-scale software systems [2]. In clouds, anomalies can be caused by operation errors [3], hardware/software failures [4], resource over-/under-provisioning [5], [6] and so on. Given the ever-increasing scale coupled with the increasing complexity of software, applications, and workload patterns, anomaly detection must operate automatically at runtime, and especially requires no intrusive modification to system monitoring and sampling in public clouds like Amazon EC2 [7].

Cloud computing has been broadly adopted for data analytics for public service [8]–[10]. In a public cloud, developers and system administrators can collect and track metrics by using cloud facilities like Amazon CloudWatch [7], which provides monitoring for AWS cloud resources and applications, and sampling customer-defined metrics. Although it is possible to gain insight for the smoothness and performance of applications, it is hard to automatically detect anomalies with only the CloudWatch data in complex system environments, because of the limitation of CloudWatch as analysed and shown in the following. Moreover, since users cannot arbitrarily access the system information in public clouds, the anomaly detection at user end has to be non-intrusive.

DevOps is a cross-disciplinary community of practice dedicated to the study of building, evolving and operating rapidly-changing resilient systems at scale [11], [12]. In DevOps, continuous delivery/deployment is an important technique to maintain high availability and operability, but non-intrusive anomaly detection is challenged by continuous delivery/deployment. For example, this kind of automated DevOps tasks will trigger various sporadic operations (e.g. upgrade, reconfiguration, redeployment or backups) that will impact the execution of a running application. In addition, continuous deployment practices make such sporadic operations much more frequent.

In this paper, for tenants and DevOps practitioners in public clouds, we propose an anomaly detection, which is designed for public cloud users to deal with the case that the impacts from DevOps operations and anomalies on the metrics are same or similar. To be more specific, we report our anomaly detection on a successful public cloud, Amazon Web Service (AWS), and a representative DevOps operation, rolling upgrade. In this context, some implementations of rolling upgrade, e.g. Asgard [13], upgrade software installed in VM instances by simply killing instances in an old version and restarting those in a new version. While, the changes of performance metrics due to rolling upgrade are almost same or similar to the changes caused by instance-level anomalies.

Our detection collects the log information of running operations and applications, and then uses the information to indicate the run-time system environments

Daniel Sun, Min Fu, and Liming Zhu are National ICT Australia, and School of Computer Science and Engineering, University of New South Wales, Australia, e-mail: first name.lastname@nicta.com.au. Guoqiang Li is with School of Software, Shanghai Jiao Tong University, P.R.C., e-mail: li.g@sjtu.edu.cn. Qinghua Lu is with College of Computer and Communication Engineering, University of Petroleum, Qingdao, P.R.C., e-mail: dr.qinghua.lu@gmail.com.

for choosing a statistical model, which has been trained beforehand. We analyse the metrics and their monitor data provided by CloudWatch and then choose Support Vector Machine (SVM) to train the models under different system environments. At runtime, newly sampled metric data are fed to a classifier that has been indicated by the log information. The experimental results show that even if the log information can help choose the correct classifier, the false positive rate is too high because of the limitation of monitor data. For this sake, we further improve the anomaly detection by introducing moving windows. Moreover, in order to tackle the most difficult case: An operation impacts on the system in the exactly same way as anomalies, we introduce moving average over the windows and the entropies of metrics as additional features to SVM. The advantage of our technique is that we only require the data and logs provided officially by cloud facilities to detect underlying anomalies effectively. Our contributions are listed as follows.

1 We train multiple classifiers by using SVM for various system and software environments that are indicated by the log information of operations and applications.
2 During detection, the logs retrieved at runtime are used to select the exact classifier and the metrics data are streamed into a classifier for anomaly detection.
3 After an analysis on the data provided by CloudWatch, we adopt moving window, and moving average and entropy features for both training and detection.

The experimental results show that our non-intrusive anomaly detection can effectively detect anomalies. The accuracy, the precision, and the recall can reach up to more than 90%. The false positive rate can be as low as 10%. These results for non-intrusive detection have been much better than threshold-based alarming, which does not work at all during DevOps operations. To the best of our knowledge, this is the first work addressing the non-intrusive anomaly detection during DevOps operations. Note that, our detection aims at the anomalies that impact on performance metrics, e.g. a lost or stacked instance that impacts on CPU utilisation, other than those anomalies that can be easily detected by other methods. In this paper, we only detect whether or not there are any anomalies, but we do not provide the diagnosis for identifying what anomaly has been detected.

The rest of this paper is structured as follows. In Section 2, we review related work. In Section 3, we introduce the monitored metrics, the logs, and a sporadic operation, rolling upgrade including its step-wise information. Then a basic approach of learning and streaming is shown in Section 4. In Section 5 we make a significant improvement after a brief analysis on the metrics. In Section 6, we present our experiments and the results. Section 7 concludes this paper and justifies our contributions.

2 Related Work

Threshold-based methods are pervasively adopted in industrial products [7], [14]–[18]. They firstly set up upper/lower bounds for each metric. Those threshold values come from performance constraints or predictions. Whenever any of the metric observation violates a threshold limit, an alarm of anomaly is triggered. Although thresholds can be set dynamically, threshold-based approaches may generate too many false positives or negatives in dynamic and complicated cloud environments, especially during DevOps operations.

Anomaly detection using cloud metrics has been shown in [2], [19]–[22], in which with statistical techniques performance models are built for abnormal behaviour and flag deviations as anomalies. Correlation-based methods have been proposed to capture performance invariants, but they are expensive to learn, and require large training data, especially for non-linear correlations. It has been shown that an ensemble of models to address variations needs to change with time in software or hardware updates [2]. In [22] an application change is identified by using two different models for a given time period and this method leads to higher accuracy. For sporadic operations, simple statistical metrics are not sufficient to characterise and distinguish the data generated by operations, especially when the impact of operations is very similar to the impact of failures. In this paper, we retrieve log information of operations and applications, and then use the information to help detection.

Anomaly detection has been recognised as a typical machine learning classification problem. Several existing machine learning methods, such as artificial neural network, decision tree, and SVM, have been employed to solve the classification problem [23]. For the case of online detection, streaming-based anomaly detection is often adopted. The monitored data streams are analysed through the techniques of time series analysis and machine learning [24]–[26]. However, this class of methods require that the streaming data must be accurate in real-time, and this is not realistic in public clouds. In this paper, we learn and predict from the streaming data by using moving average and entropy over a buffer of streaming data to detect anomalies within a time period.

Many stream processing frameworks focus on dealing with data sources continuously producing data. For example, Spark [27] has a good streaming support integrated, which supports the building of real time predictive analytics services. Spark Streaming [28] is an extension to the Spark core, which provides fast and scalable streaming data processing, where it integrates batch processing in streaming processing. Apache Storm [29] is another distributed computing framework for real time data processing. For large-scale systems and applications, especially the case that machine learning models

need to be trained online, these streaming processing systems are applicable.

3 Background

3.1 Data sources

We do not optimistically assume that runtime data can be arbitrarily accessible in public clouds. In practice, two sources of data are usually provided and available in most production systems: monitored metrics and logs.

3.1.1 Monitored Metrics

Performance metrics are sampled to be time series data, which are numeric values or character strings indicating system performance and states. In AWS, CloudWatch[1] provides 54 metrics in more than 40 types, including simple statistics of each metric, such as average, minimal, maximum, and sum. The data are sampled and provided in JSON file, which is accessible via an API.

The sampled data are not real-time information and typically aggregated per-minute from individual virtual machine instances. There could be a significant time delay between an occurrence of anomalies and an appearance of abnormal value. Moreover, abnormal values could last for a period of time even after the underlying anomalies have been recovered.

3.1.2 Logs

The sources of logs include applications, web servers, database systems and operating systems. DevOps operations themselves also produce logs. The application and operation information can be retrieved automatically from the log lines by distributed log processors deployed in VM instances in clouds. Some information can be obtained instantly such as start time and end time of an application or an operation, while some information has to be retrieved through log analysis and mining.

3.2 DevOps Operations and Monitoring

DevOps operations such as upgrade, redeployment, snapshotting and on-demand scaling are often sporadic: Some are triggered by ad hoc bug fixing and feature delivery, while others are triggered by periodic maintenance activities. It has been shown that even periodic maintenance activities may not be synchronised and predictable [30]. Some operations have significant impacts on metrics. For example, during upgrading an individual instance may join and leave, or be rebooted so as to cause metrics value exceptions, which could be misinterpreted as alarms if the detection is simply based on some thresholds. In the mean time, true anomalies during the operations may be hard to detect.

1. Here, CloudWatch refers to its latest version and functions until December 2014.

3.3 Rolling Upgrade

Rolling upgrade is a representative DevOps operation, which is critical for continuous software delivery and high frequency release [12]. An application is usually deployed in a cloud onto a collection of virtual machine instances. In a typical rolling upgrade process [31]–[34], a small number of instances are upgraded from an old version to a new one at a time. This process is repeated in a wave rolling until all instances are in the new version. In clouds, the upgrade can be done by simply replacing old virtual machine instances with newly provisioned instances. The advantage of rolling upgrade is that availability does not degrade too much, since there are only a small number of instances out of service. The time of replacing a single instance in AWS is usually in the order of minutes. For hundreds of instances, a rolling upgrade will take hours or longer.

Rolling upgrade has been widely used for upgrading distributed software, but there are many different implementations in different systems. In this paper, we refer to the Asgard, which is Netflix's customised management console on top of the AWS infrastructure. Fig. 1 shows a rolling upgrade procedure of Asgard [35]. During a rolling upgrade, anomaly detection is challenging. First, a number of instances are being taken out of service (often through killing them simply) legitimately causing naive threshold-based alarm systems to raise false alarms. Second, a true fault of an instance during this period exhibits similar behaviour and has similar impacts as killing an instance. Finally, other simultaneous applications and varying workload may confound any detection mechanisms. For a specific system that provides stable services, the patterns of workload, and DevOps operations are usually countable. This gives us an opportunity to detect anomalies more effectively than traditional methods, since we can always acquire system status from logs.

3.4 Process and Step-wise Information from Logs

The process information refers to the process id, instance id, start and end times of operations and applications, and their characteristics. Inside a process, there is some information in finer grain, but the information is not explicit in logs. Our previous work [35] automatically mined the procedure/step-wise information from the logs produced by rolling upgrade. The steps are dependant and sequential such that they can be organised into a model, named process model. Along with the process model, a set of regular expressions are associated with each of the steps, which can be used by the corresponding log processor to match the log lines and extract the process context information from the log lines at runtime. Thus, we can know the exact start and end times, and progression of an operation. For other operations and applications, we can also extract at least the start and end times from log lines. For example, we can know if a rolling upgrade is running or not and at

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2016.2520883, IEEE Transactions on Emerging Topics in Computing

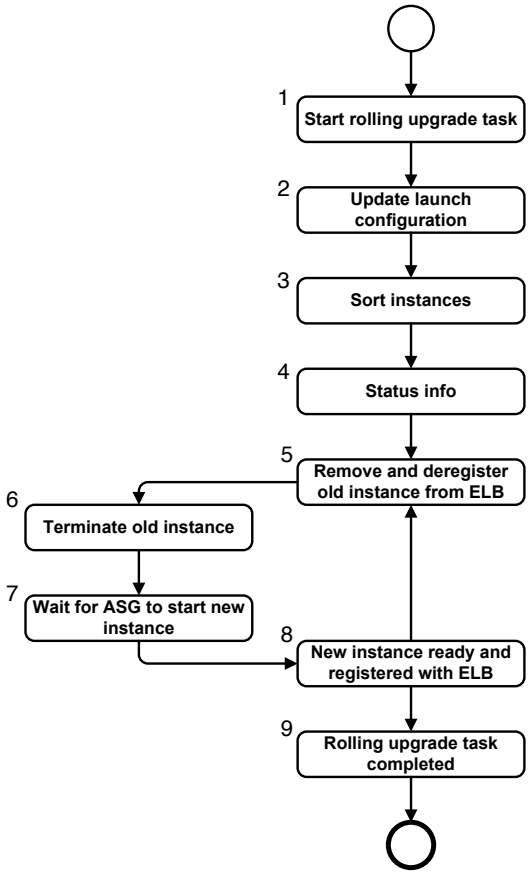


Fig. 1: Procedure of rolling upgrade by Asgard.

which step the process is. When the process is between Step 5 and Step 8 in Fig. 1, the metrics will be impacted by rolling upgrade. When a log line indicating Step 9 appears, we can know that there will be no more impact from rolling upgrade. Similarly, we can also know which applications are running at the same time.

4 Basic Learning and Streaming

4.1 Features

As we have known in Section 3.1.1, there could be 54 features from CloudWatch metrics in 40 types. However, not all of them are useful, since some metrics are not relevant to instances and cannot be impacted by anomalies. We are only interested in group-level (ASG in our scenarios) metrics and instance-level metrics, because these metrics can be impacted by anomalies. The metrics of CPU and network are always available in most public clouds and sensitive to anomalies. Hence, CPUUtilisation, NetworkIn, and NetworkOut are all adopted as features. Each of them has its maximum value, minimum value, and average value per minute. Thus, there are 9 features. We also examined other metrics that are valued in real numbers, but their weights in the model trained by SVM are very small and hence we only chose the 9 features. We also took into account two types of

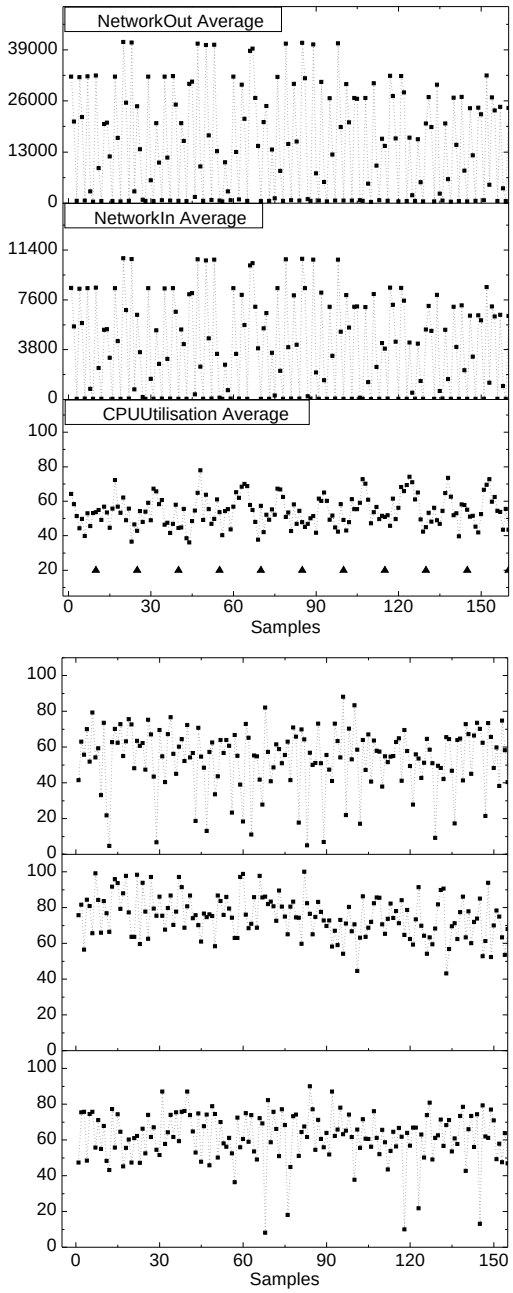


Fig. 2: Examples of metrics data. The top figure shows only three different metrics for a mix of rolling upgrade, background applications, and anomalies. The bottom one shows the different CPU utilisations for only background applications, only rolling upgrade, and idle system. A number i along X-axis indicates the ith sample. Y-axis is the value of a metric. For CPUUtilisation, it is CPU usage and for Network it is the number of bytes.

background applications: CPU-intensive and Network-intensive applications. Some examples are shown in Fig. 2. The triangles represent the anomalies. Since the top three figures share the same horizontal axis and the anomalies are among the samples, the triangles are only shown once along the horizontal axis. As we can see it is

hard to find a pattern that characterises anomalies in the metrics. The bottom three figures show different system environments, only rolling upgrade, heavy application workload, and light application workload. For the metric of CPUUtilisation, we can clearly see the difference in the magnitude under different system environments. This phenomenon testifies that using the information from the logs can simply distinguish the metrics data from different environments.

We also plot the three exemplar metrics into a scatter matrix in Fig. 3. It is easy to observe that there is no obvious correlation except between NetworkIn and NetworkOut. When anomalies appear, only a very small number of requests will be lost and this can result in NetworkIn being greater than NetworkOut slightly. After the other healthy instances start to receive requests, the data of two metrics will be fully linearly related again. Moreover, the aggregated values almost make this change invisible. Since the change of the correlation is very small, it is impossible to utilise this phenomenon to detect anomalies. Although we cannot show all the plots due to space limit, one can easily conclude that the aggregated metrics do not have pair-wise correlation of metrics in different types. As a result, the correlation among the metrics in the same type is not helpful, while the metrics in different types do not have obvious correlation. This implies that simple statistical detection based on correlations cannot work well.

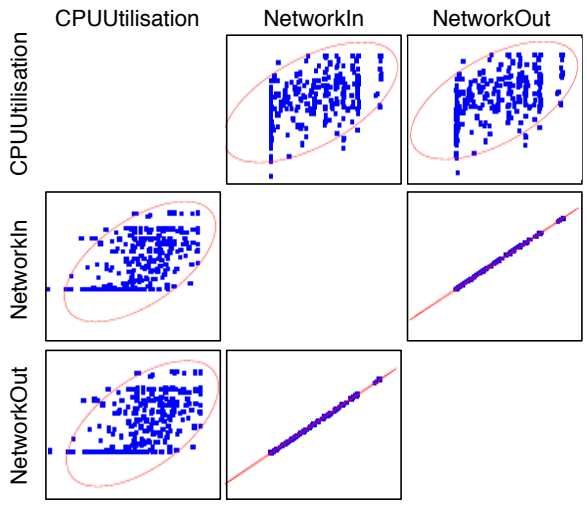


Fig. 3: A scatter matrix.

Note that, there is a metric named *GroupInServiceInstances*, which indicates the number of physically healthy instances in ASG. It is indeed a useful indicator for the anomalies that cause detectable unhealthy instances, although this metric is not real-time either. But for software and service anomalies, this metric is misleading, since the anomalies only disable software and service but the instances are still healthy. Thus, the anomalies are not detectable to EC2 health checker. Hence, we cannot use this metric as a feature because it may cause many unnecessary false negatives.

4.2 Algorithm and Training

Since there is no obvious pair-wise correlation between different types of metrics, and the correlation between the metrics in the same types is not helpful for detection, we initially have to consider all the features of metrics, although we finally lock on 9 original features. Thus all the features form a high-dimensional feature space, for which Support Vector Machine (SVM) has been known as a well suitable machine learning algorithm. There are several existing tools that can be employed for training, but we only use LibSVM toolbox in Matlab in this paper.

Fig. 2 has shown that different software and system environments generate different signals of metrics. It is possible to use the information from logs as new features, but most of the information is binary which indicates whether or not an operation or application is running, and the binary information is rigid and may cause significant overfitting. Hence, we use the binary log information as the indicators, each of which corresponds to a specific classifier that has been trained from SVM. Hence, we need to collect the monitor data from separated scenarios to train different classifiers. At runtime, by monitoring the log lines we can select a classifier. For the purpose of clearly presenting our idea, the implementation in this paper covers 4 cases, clear background (idle system), only background applications, only rolling upgrade, and a mixture of applications and rolling upgrade, all of which suffer from the anomalies, i.e., injected faults.

4.3 Data Streaming and Detection

As we have introduced in Section 3.1.1, the sampled data can be fetched through AWS API. There are some software tools that support data streaming and queueing, and can be deployed as a streaming data service as summarised in Section 2. In this paper, we built our own data streaming service by simply calling the API. After an instance of data is fed to a selected classifier, the output of this classifier tells if there is any anomaly in the current system. The selection is determined by the system state that is indicated by log lines, and this implies that the log information should be retrieved and synchronised with the monitored metrics data. We also implemented distributed log processors for that purpose. The implementation of detection is shown in Fig. 4.

We will see in Section 6 that the prediction on each instance of data is too poor to be useful in practice. Our experiments on AWS testifies that simply using the monitored metrics data sampled by CloudWatch to train SVM models and detect anomalies does not work so well, even if the log information is supplied.

5 Improvement

Because detecting anomalies on data points cannot achieve satisfying detection results, we must improve

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2016.2520883, IEEE Transactions on Emerging Topics in Computing

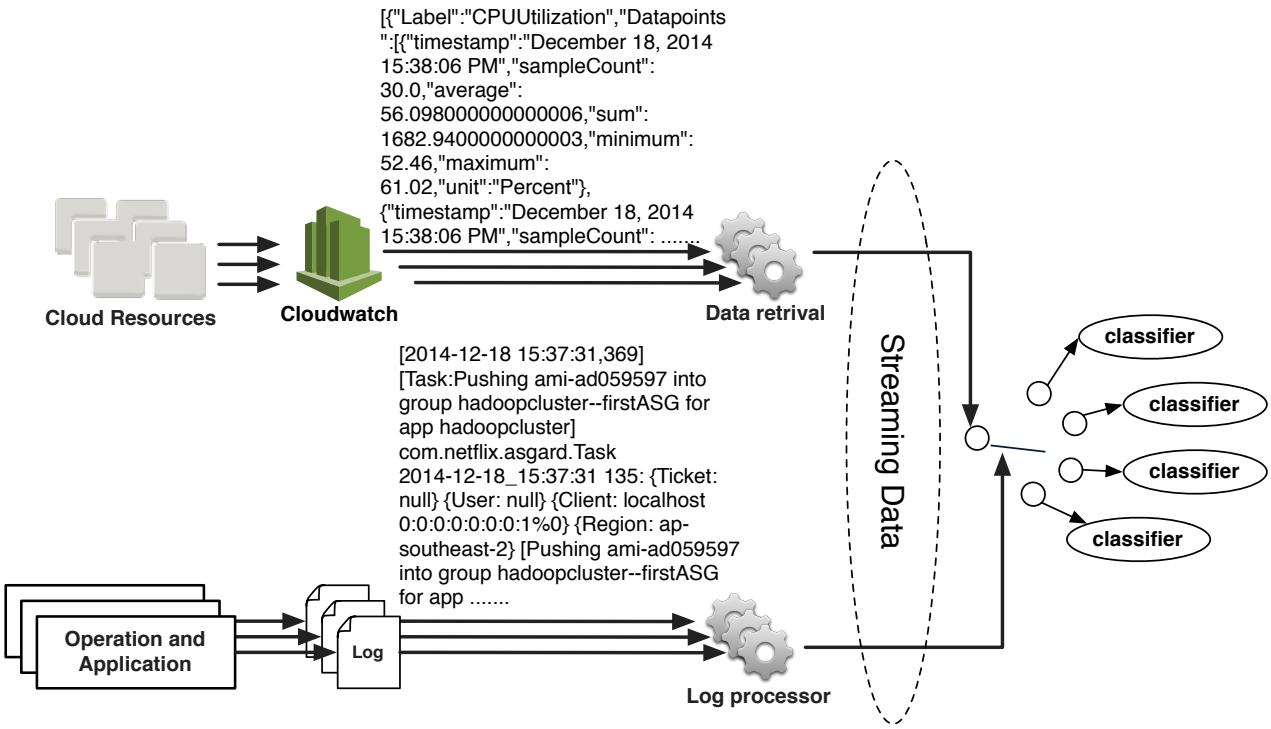


Fig. 4: Implementation of data streaming and detection.

the above technique. In this section, we first analyse why detecting on data points is not acceptable and then we introduce our improvement, in which we train the classifiers and detect anomalies on a predefined time interval, which is called *window*. Thus, the data fed to the classifiers are sampled by CloudWatch every w minutes and w is the size of window. In this paper, we only adopt fixed windows rather than dynamic batch mode processing of streaming data e.g. in [36].

5.1 Information from The Metrics

As we have introduced about CloudWatch sampling and data aggregation in Section 3.1.1, it is not realistic to detect anomalies in real-time and inherently on data points, because we cannot arbitrarily monitor a system and access infrastructure data. AWS has been a successful public cloud platform, and hence CloudWatch represents the current technology of infrastructure data service in public clouds. That is, user-accessible monitored metrics data and logs are hardly applicable by simply hiring machine learning for anomaly detection in DevOps.

Although real-time anomaly detection has become intractable, we can provide an efficient detection over a longer window size. On the longer scale of time, we can have sufficient information for anomaly detection. For example, there will be 5 data instances for each feature if we set the window size to be 5. It is always true that more information results in better understanding of a system. If CloudWatch could provide aggregated monitored data within seconds, we could detect anomalies exactly every minute.

In the following, we look into the implication of statistics and information theory for the adoption of moving average and entropy as new features.

5.2 Moving Average

In statistics, there are many ways to obtain data characteristics from the data within a window. With respect to the implementation of our data streaming and learning, we choose *simple moving average* (SMA), which is the unweighted mean of the w numbers in a window. Let d_t be the number at the t^{th} minute and then d_{t+w-1} is the number at the $(t+w-1)^{th}$ minute from CloudWatch. Consequently SMA is exactly

$$SMA = \frac{\sum_{i=0}^{w-1} d_{t+i}}{w}. \quad (1)$$

SMA captures the average change of data in different windows, but it looses the information of peaks and valleys in the original signal. Fortunately, CloudWatch has the maximum, the minimum, and the average of original signal separately. Thus, we compute SMA for each metric, and then we use all SMAs as the features in additional to the original monitored data.

5.3 Entropy

Only moving average is not sufficient to reflect the impacts of anomalies. The metrics can be impacted by underlying infrastructure, running virtual machines, applications, operations, and anomalies. The value of a metric is usually random. Each metric can be treated as

a random variable. As shown in the following, when anomalies impact on the same system, the randomness of a metric should be likely higher than before anomalies happen, provided that anomalies are independent of the other factors impacting the metrics.

In information theory, Shannon's Entropy [37] is a widely used measurement that captures the degree of dispersal or concentration of random variable distributions [26], [38]. For a discrete[2] random variable X with possible values $x_1, x_2..., x_n$, its entropy is:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log P(x_i), \tag{2}$$

where $P(x_i)$ is the probability mass function of outcome x_i. $logP(x_i)$ is called *surprisal* or *self-information* of x_i. In the following, we analyse why and how entropy can be a good feature.

Sampling a metric is equal to taking a value of corresponding random variable. For the metrics of CPU utilisation and network, even there is no running application and operation, the values of the metrics are not always zero and show a certain randomness, because at least the virtual machines need to maintain running status and EC2 heath checker monitors them every a time period. The values of the metrics are all aggregated data, to which each instance independently contribute. Generally speaking, in many cases we can treat the impact of an anomaly on a metric as an independent random variable. In the following, we assume this independency, and then discuss and justify the assumption.

In the literature, there has been rich theory for us to explore if entropy can be a feature in our detection. Let X_M denote the random variable that represents a metric, and let X_a be the impact from an anomaly onto X_M. If X_M and X_a are independent, when the anomaly is impacting the metric, the metric value should be $X_M + X_a$. Note that, the impact is always positive, because when an anomaly takes an instance out of the service the other instances have to process more workload and thus the metrics like CPUUtilisation, NetworkIn and NetworkOut on instances must be increased. Then, we are interested in the difference between $H(X_M+X_a)$ and $H(X_M)$.

Theorem 1 *When X_M and X_a are independent, we always have $H(X_M + X_a) \geq H(X_M)$.*

Proof This can be proved from the property of *Entropy of a sum* [37]: Let X and Y be independent random variables. Let $Z = X + Y$. $H(Z) \geq \max\{H(X), H(Y)\}$. Then it is obvious to conclude it. ∎

Entropy of a sum is an elementary property of entropy and an exercise in [37]. The proof of this property has many versions in the literature such that we omit the proof in this paper. From the theorem, we can easily conclude the following corollary.

2. because the metrics are sampled always in discrete time.

Corollary 2 *New anomalies will likely increase the entropy of metric value; If recoveries can dismiss completely the impact of anomalies on the metric, the entropy will likely be decreased, only if all of them are independent of other factors that impact the metric too, and the other factors are stable.*

Proof X_M represents the random value of a metric, which is a synthesis of many factors. Let X_a denote a random impact of an anomaly on the metric value. In terms of Theorem 1, for a new anomaly, the entropy of $X_M + X_a$ will remain the same or be increased. When this anomaly is recovered, as recoveries eliminate the impacts of anomalies on the metrics, the random value of the metric is still X_M. Thus, the entropy of the metric may decrease or keep in the same. Note that the independence is needed to guarantee Theorem 1 such that it is also necessary here. Theorem 1 also implies that X_M should not be changed, i.e. the other factors are stable in a probabilistic sense. ∎

Only if anomalies are independent and can be fully recovered as requested by Corollary 2, the entropy of the metric should show certain monotonicity. As we know that X_M is the synthesis of many factors, let X_B be the metric value without anomalies, then $X_M = X_B + X_{a1} + X_{a2} + \cdots$. The above theorem and corollary are true, only if X_B is fixed. X_B represents the random impact of workload, applications, and the background except anomalies. When the software and system environment changes, the above reasoning will not be true any longer. Fortunately, we have the log information that switches the classifiers to different environments. Only if the switch works well, are Theorem 1 and Corollary 2 true for each classifier in the corresponding environment. Besides the relation, Theorem 3 implies that the entropy cannot always increase and must be bounded in terms of the number of possible metric values. This theorem can help us compute the approximation of entropy in the following.

Theorem 3 (Theorem 2.6.4 [37]) *Let X be a random variable and $|\mathcal{X}|$ be the number of elements in the range of X. $H(X) \leq \log |\mathcal{X}|$.*

From the above theoretical analysis, we know that more anomalies will increase entropy such that the entropy of a metric is a useful feature for SVM to measure if anomalies have impacted the randomness of a metric. However, it is too optimistic to assume that we can know the probabilities required in (2). Because we only have the monitored metrics data, in this paper we choose the approximation of entropy.

For each metric, we can know the maximum value V_{max} and the minimum value V_{min}. From V_{min} to V_{max}, we partition the real line into a number of segments and the number should be as big as possible in terms of Theorem 3. In practice, we set the number just bigger than the window size, because there are at most w values in a window. The total number of the segments is S and a segment is s_i. For each s_i, the minimum of metric value

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2016.2520883, IEEE Transactions on Emerging Topics in Computing

is v^i_{min} and the maximum is v^i_{max}. Given w metric value in a window, we count on the occurrence of s_i, that is, for a metric value v_i, if $v^i_{min} \leq v_i \leq v^i_{max}$, s_i appears once. Let n_i be the frequency count of s_i, then we can calculate the entropy of discrete random variable s in a window as follows:

$$H(s) = -\sum_{i=1}^{S} \frac{n_i}{w} \log \frac{n_i}{w}, \quad (3)$$

The entropy calculated from the frequency count is used as a feature for SVM. Eq. 3 is a kind of histogram estimator, which is biased [37]. There are other useful probability density function (pdf) estimation methods. For example, Gaussian mixture modelling (GMM), where the expectation maximisation (EM) algorithm is used to find a maximum-likelihood estimate that approximate the data pdf. In this paper, we will not go so far, since this paper focuses on the methodology that we propose for anomaly detection.

To sum up, we use the entropies of metrics defined and calculated above as additional features for training and detection. Thus, the total number of features including the features of moving average and entropy becomes 27. Note that, the change of entropy implies the change of randomness due to anomalies rather than the magnitude. An example of SMA and entropy features for the metric CPUUtilisation is shown in Fig. 5. We also make some scatter plots, from which we can see the correlation between different features including the entropies in Fig. 6, and it is visible that the correlation between the features is stronger than in Fig. 3. For example, in Fig. 3 the correlation between CPUUtilisation and NetworkIn is much weaker than that between CPUUtilisation and the entropy of NetworkIn (H of NetIn) in Fig. 6.

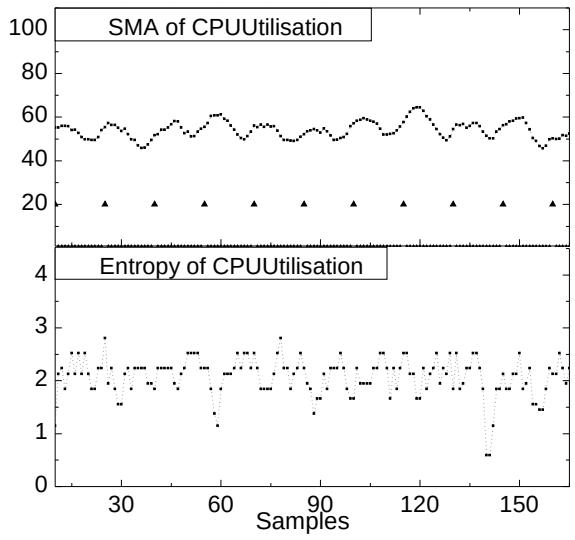


Fig. 5: New features.

5.4 Discussion

In this paper, as shown above, we have made some assumptions. In this section, we discuss and justify the assumptions.

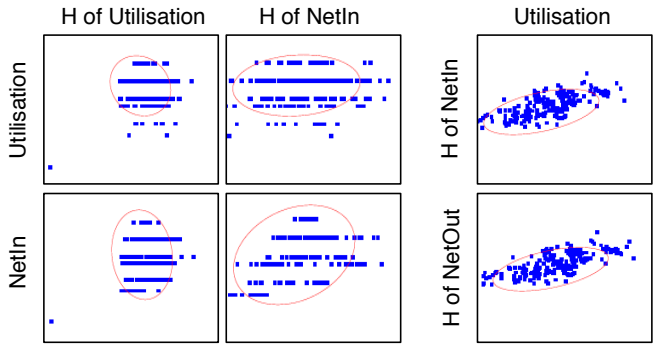


Fig. 6: Scattered Data of features. H refers to entropy, NetIn is NetworkIn, and CPUUtilisation is shorted as just Utilisation.

We have assumed that all the instances are providing a service independently. There are many systems and applications in this kind, e.g. a server farm for web services. Thus, anomalies impact the instance level metrics and the aggregated metrics independently. Our method does not suit the case that the instances are not independent. For example, there exists a critical instance. When this critical instance fails, the whole system will stop. Then, some metrics such as CPUUtilisation, NetworkIn and NetworkOut will simultaneously decrease to their minimum. For this case, the symptom of anomalies is so obvious that it does not need an effort of detection.

We also have assumed that a recovery can completely remove the impact of an anomaly on a metric. From our analysis, only in this case will anomalies increase metric entropy such that entropy can be used as a feature. For independent instances, this assumption is fair in clouds because to recover an anomaly it is only necessary to replace the instance where the anomaly is.

The system design as shown in Fig. 4 requires that the system statuses can be classified and indicated by logs. We assume that the number of system statuses is not too big to train the classifiers. This assumption is realistic for most web applications deployed in public clouds.

6 Empirical Studies

In order to evaluate our proposed anomaly detection, a series of experiments have been performed in AWS. In this section, we report the experimental results in terms of the accuracy, the precision and recall.

6.1 Experiment Deployment and Setting

A content management system is deployed to an ASG (Auto Scaling Group) containing 8 identical EC2 instances[3] that can be upgraded in a rolling upgrade

3. In order to run our experiments on a real public cloud, the experiments need financial support and hence the number of instances cannot be very big, since we need to run the long term experiments for data collection in different cases.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2016.2520883, IEEE Transactions on Emerging Topics in Computing

fashion. We used Siege, an http load testing and benchmarking utility, to generate user requests as a background workload for our experiments. We performed rolling upgrade with Asgard on the EC2 instances through heavily-baked upgrading approach, which uses pre-baked images to replace existing VM instances. To simulate a complex ecosystem, we ran another two background applications, a CPU-intensive application (compressing data) and a Network-intensive (receiving and responding data requests) application. Both of them also produce logs. A fault injection program is running on the side to inject EC2 instance faults that can halt instances. An anomaly in AWS (micro/small instances, heavily-baked images, and using EC2 health checker) can last even more than 10 minutes. Hence, we create the anomalies with an interval of 15 minutes, that is, we do not allow too many overlaps among anomalies, since more overlapped anomalies are easier to detect.

A particular system and application context may have a bias to some factors, for example the number and the frequency of anomalies, the overlap among background applications and operations, and the types of anomalies and so on. In order to cover most possible cases, the setting of our experiments tries to mimic possible system environments and provides different number of fault injections.

We intertwine the two applications and rolling upgrade operation in different overlaps. Instance faults are injected randomly. The granularity, G, of the rolling upgrade is the number of instances that can be upgraded together. We set the granularity to be 1 or 2. Eventually, we have many groups of experiments, each of which lasts around 2000 minutes. A half group of data are for training the classifiers, and the other half is for verifying the detection. Note that the experiment setting twists rolling upgrade, applications, and fault injections to be more complicated than most cases in reality. If we can achieve satisfactory results on this, we can deal with most real cases easily. Once again, our detection is designed to deal with the same or similar impacts of DevOps operations and anomalies on the metrics. We calculated the accuracy, the precision and the recall from the experimental results, and compared different experiment settings. P is the number of all positives and N is the number of all negatives. TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives. As formally defined in the literature,

$$Accuracy = \frac{TP + TN}{P + N}, \tag{4}$$

$$Precision = \frac{TP}{TP + FP}, \tag{5}$$

and

$$Recall = \frac{TP}{TP + FN}. \tag{6}$$

Note that *false positive rate* is just $1 - Precision$.

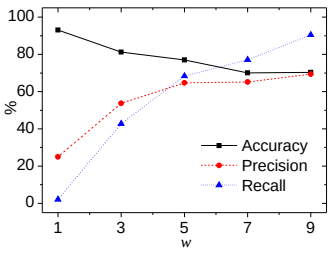


Fig. 7: Idle system.

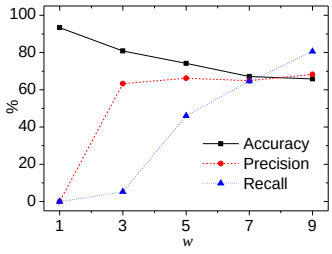


Fig. 8: Only rolling upgrade ($G = 1$).

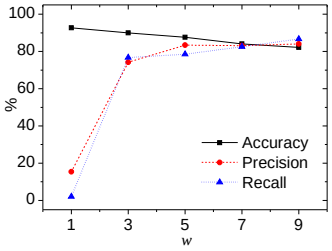


Fig. 9: Only applications.

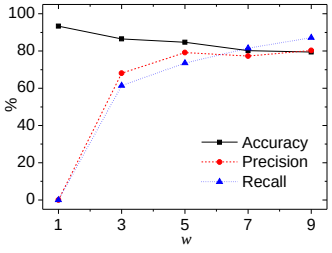


Fig. 10: Applications and rolling upgrade ($G = 2$).

6.2 Experiment Results

In all the experiments, $w = 1$ is the basic detection without SMA and entropy features in Section 4. For w greater than 1, the detection uses SMA and entropy features. When $w > 1$, the training and the detection are for the windows. Thus we need to label windows during training process. For example, when an anomaly happens in the time $t = 2$, for $w = 3$ the windows $[0, 2]$, $[1, 3]$, and $[2, 4]$ are all labeled to be with anomalies.

As we can observe, the precision and the recall at $w = 1$ are very low, while the accuracy at $w = 1$ is high. Because we only injected an anomaly every 15 minutes and hence P/N is small (CloudWatch samples the metrics every minute), a high accuracy implies that lots of true negatives are predicted. Then we can know that few true positives can be detected. As we have analysed above, it is not realistic to detect anomalies on 1 minute scale in this paper. The results at $w = 3$ are not good either, because the window size is short and there is no sufficient information for the detection even if we have adopted SMA and entropy features. The situation becomes much better when the window size becomes greater than 5. In nearly all figures, the accuracy keeps deceasing. We use the following example to explain this phenomenon: Let 0 represent the normal and 1 the abnormal. Given a series of streaming events marked by 1 or 0, e.g. 00100100, without using window (actually $w = 1$), if we loose all "1"s, the accuracy is 75%, and with $w = 3$ the streaming becomes 111111, if we also loose two "1"s, the accuracy is 66.67%, and while the precision and recall are dramatically increased. Under such a condition, only if we loose the first '1", we miss a real anomaly, and if we loose the others, we can still correctly predict anomalies during the time periods.

Fig. 7-10 are the results from completely separated

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2016.2520883, IEEE Transactions on Emerging Topics in Computing

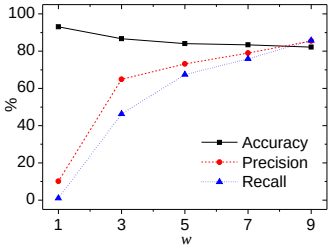


Fig. 11: Mix of all with log information ($G = 2$).

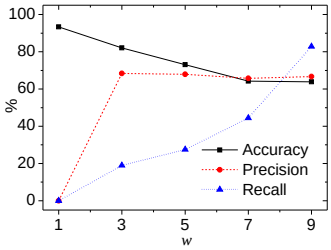


Fig. 12: Mix of all without log information ($G = 2$).

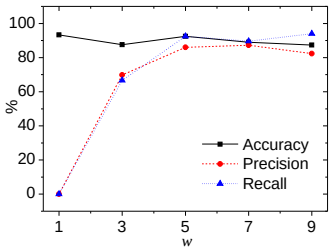


Fig. 13: Only applications.

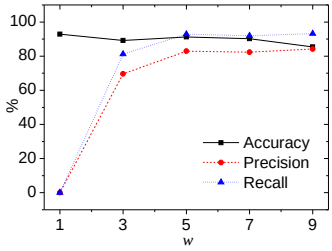


Fig. 14: Applications and rolling upgrade ($G = 1$).

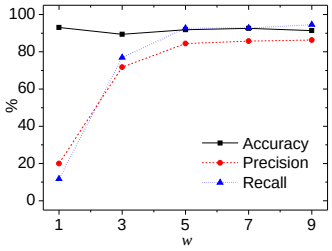


Fig. 15: Mix of all without log information ($G = 1$).

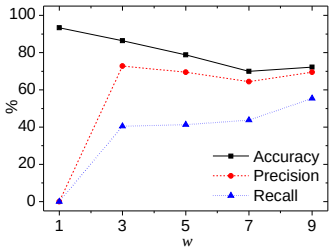


Fig. 16: Mix of all without log information ($G = 1$).

scenarios, where the system is idle, runs only rolling upgrade, runs only applications, or runs applications and rolling upgrade. We test our anomaly detection in each scenario and thus there is no need to switch the classifiers. In Fig. 7 and Fig. 8, the detection is not satisfying because when there is no workload, the performance metrics can be impacted by anomalies slightly.[4] A rolling upgrade with the granularity of 1 impacts the system as the anomalies, which do not appear too frequently as mentioned above. Because the impact of loosing VM instances is hardly observed if there is no workload, we can see that rolling upgrade leads to more false negatives, i.e. low recall, for the detection recognises the impacts from anomalies as those from rolling upgrade.

In Fig. 9 and Fig. 10 the detection can achieve meaningful precision and recall after $w = 5$, because when there is workload, an anomaly impacts the metrics heavily. Upgrading one instance at a time is almost the same as an anomaly disabling an instance, while upgrading more will make the detection harder since we statistically

4. Load-balancer can move workload from problematic VMs to healthy ones, and then the metrics such as CPUUtilisation can be increased due to anomalies. If there is no workload, the impact on those metrics is hardly observed.

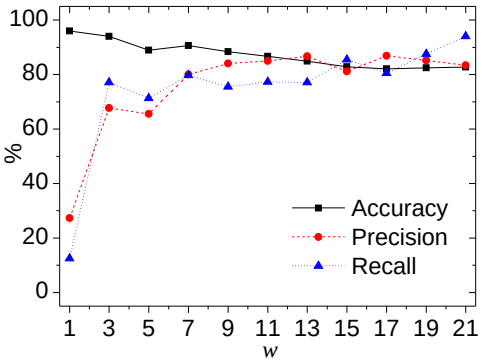


Fig. 17: Longer window sizes ($G = 2$).

allow only one anomaly. This can be observed by comparing Fig. 10 to Fig. 14, smaller granularity resulting in better detection. For the normal scenario, a mix of the different cases above, the log can indicate the cases and then the best classifier can be selected. With the log indicated switch, the best accuracy, the precision, and the recall reach more than 90% for the granularity of 1 as shown in Fig.**??**, and for granularity of 2, they are around 80% as shown in Fig.11. The false positive rate can be reduced to 10% in the best case. Generally speaking, the results depend on how the system, the rolling upgrade, and the application are running. In most cases similar to our settings, i.e. a system with active applications, rolling upgrade, and little idle time, our detection is effective usually.

We also test the detection without the log indicator. The results are in Fig. 12 and Fig. 16. It is easily observable that the precision and the recall are too poor to be useful in practice. From all the above experiments, one may conclude that if the length of window is sufficiently long the detection performance will become better and better. This is true for frequently appearing anomalies. However, longer windows imply longer detection delay. If we can achieve a satisfying detection within a short window, a shorter window is always preferable. When anomalies appear not so often, longer windows may not be beneficial and the gain is marginal as shown in Fig. 17, in which the average interval between anomalies is 30 minutes and the granularity is 2.

There are also some cases, where our detection cannot work well such as when many metrics data reach their maximum. One example is the heavy workload, which makes CPU Utilisation reach its maximum. In this case, the metric of CPUUtilisation will not be effective any longer, and the detection has to rely on other metrics. Thus, the results cannot be as good as the system environment of normal workload. We will not show more stress tests for our detection in this paper.

7 Conclusion

In this paper, we propose an approach that makes use of log information to select different classifiers, which are

trained with monitored metrics data via SVM, in public clouds during DevOps operations. We specify our work with a successful public cloud, AWS, and a representative DevOps operation, rolling upgrade. Through both our analysis on the metrics data and the experiments, we can conclude that non-intrusive anomaly detection for data points, i.e. detection for 1 minute interval, is not realistic in AWS during DevOps. Hence, we make an improvement on the basic streaming and learning approach by adopting simple moving average and entropy features on longer detection windows. The experiments show that our proposed approach can achieve the accuracy, the precision and the recall no less than 90% with the low false positive rate around 10% for 9-minute window. Hence, we conclude that without low-level and real-time information, only using the data provided by cloud facilities to detect anomalies non-intrusively can be achieved in practice, but the price is that the detection is also not real-time and the delay longer than 3 minutes is inevitable for DevOps operations in public clouds.

Provided that a cloud-based system can be monitored and sampled more frequently and accurately, the result in this paper can be dramatically improved to be real-time. As we can know from this paper, detailed information about DevOps, for example the procedure of rolling upgrade, is very helpful for anomaly detection. Hence, our future direction on anomaly detection for DevOps is information retrieval and modelling of DevOps operations.

ACKNOWLEDGMENTS

Guoqiang Li is supported the National Natural Science Foundation of China (No. 61472240, 91318301). NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

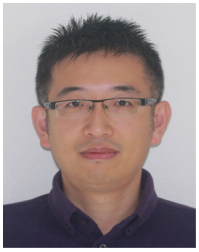
REFERENCES

[1] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection : A survey," *ACM Computing Surveys*, September 2009.
[2] S. Zhang, I. Cohen, J. Symons, and A. Fox, "Ensembles of models for automated diagnosis of system performance problems," in *The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN2005)*, Yokohama, Japan, 2005, pp. 644–653.
[3] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USENIX Association, 2003.
[4] S. Pertet and P. Narasimhan, "Causes of failure in web applications," CMU, Tech. Rep. CMU-PDL-05-109, 2005.
[5] V. Kumar, K. Schwan, S. Iyer, Y. Chen, and A. Sahai, "A state-space approach to sla based management," in *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, April 2008, pp. 192–199.
[6] V. Kumar, B. Cooper, G. Eisenhauer, and K. Schwan, "imanage: Policy-driven self-management for enterprise-scale systems," in *Middleware 2007*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 287–307.
[7] [Online]. Available: https://aws.amazon.com/
[8] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen, "G-hadoop: Mapreduce across distributed data centers for data-intensive computing," *Future Generation Compter Systems*, vol. 29, no. 3, pp. 739–750, 2013.
[9] L. Wang, H. Geng, P. Liu, K. Lu, J. Kolodziej, R. Ranjan, and A. Y. Zomaya:, "Particle swarm optimization based dictionary learning for remote sensing big data," *Knowledge-based Systems*, vol. 79, pp. 43–50, 2015.
[10] W. Song, L. Wang, R. Ranjan, J. Kolodziej, and D. Chen, "Particle swarm optimization based dictionary learning for remote sensing big data," *IEEE Systems Journal*, vol. 9, no. 2, pp. 416–426, 2015.
[11] [Online]. Available: http://theagileadmin.com/what-is-devops/
[12] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, ser. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.
[13] [Online]. Available: http://techblog.netflix.com/2012/06/asgard-web-based-cloud-management-and.html
[14] L. Wang, S. U. Khan, D. Chen, J. Kolodziej, R. Ranjan, C.-Z. Xu, and A. Y. Zomaya:, "Energy-aware parallel task scheduling in a cluster," *Future Generation Compter Systems*, vol. 29, no. 7, pp. 1661–1670, 2013.
[15] [Online]. Available: http://www.hp.com/go/sim
[16] [Online]. Available: http://www.ibm.com/tivoli
[17] [Online]. Available: http://www.nagios.org/
[18] [Online]. Available: http://www.nimsoft.com
[19] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham, "E2eprof: Automated end-to-end performance management for enterprise systems," in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, June 2007, pp. 749–758.
[20] G. Bronevetsky, I. Laguna, B. de Supinski, and S. Bagchi, "Automatic fault characterization via abnormality-enhanced classification," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, June 2012, pp. 1–12.
[21] B. Sharma, P. Jayachandran, A. Verma, and C. Das, "Cloudpd: Problem determination and diagnosis in shared dynamic clouds," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013, pp. 1–12.
[22] L. Cherkasova, K. M. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change." in *DSN*. IEEE Computer Society, 2008, pp. 452–461.
[23] R. S. Michalski, I. Bratko, and A. Bratko, Eds., *Machine Learning and Data Mining; Methods and Applications*. New York, NY, USA: John Wiley & Sons, Inc., 1998.
[24] S. Papadimitriou, J. Sun, and C. Faloutsos, "Streaming pattern discovery in multiple time-series," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05, 2005, pp. 697–708.
[25] K. Carter and W. Streilein, "Probabilistic reasoning for streaming anomaly detection," in *Statistical Signal Processing Workshop (SSP), 2012 IEEE*, Aug 2012, pp. 377–380.
[26] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online detection of utility cloud anomalies using metric distributions," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*, April 2010, pp. 96–103.
[27] [Online]. Available: https://spark.apache.org/
[28] [Online]. Available: https://spark.apache.org/streaming/
[29] [Online]. Available: https://storm.apache.org/
[30] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408794
[31] T. Dumitraş and P. Narasimhan, "Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system," in *Middleware*, 2009, pp. 18:1–18:20.
[32] V. Gramoli, L. Bass, A. Fekete, and D. Sun, "Rollup: Non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations," *IEEE Transactions on Parallel and Distributed Systems*, 2016.
[33] D. Sun, L. Bass, A. Fekete, V. Gramoli, A. Tran, S. Xu, and L. Zhu, "Quantifying failure risk of version switch for rolling upgrade on clouds," in *Proceedings of International Conference on Big Data and Cloud Computing*, 2014.
[34] D. Sun, D. Guimarans, A. Fekete, V. Gramoli, and L. Zhu, "Multi-objective optimisation for rolling upgrade allowing for failures in clouds," in *Proceedings of IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2015.

[35] X. Xu, L. Zhu, L. Bass, I. Weber, and D. Sun, "Pod-diagnosis: Error diagnosis of sporadic operations on cloud applications," in *DSN*, 2014.
[36] W. Sun, Y. Zhang, and Y. Inoguchi, "Dynamic task flow scheduling for heterogeneous distributed computing: algorithm and strategy," *IEICE Transaction on Information and Systems*, vol. 90, no. 4, pp. 736–744, 2007.
[37] T. M. Cover and J. A. Thomas, *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.
[38] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 241–252.

Daniel Sun (S'06-M'08) received his Ph.D. in information science from Japan Advanced Institute of Science and Technology (JAIST) in 2008. From 2008 to 2012, he was an assistant research manager in NEC central laboratories in Japan. In 2013 he joint National ICT Australia as a researcher. He is also a conjoint lecturer in School of Computer Science and Engineering, the University of New South Wales, Australia. His current research interests include system modelling and evaluation, algorithms and analysis, reliability, energy efficiency, and networking in parallel and distributed systems and big data.

Min Fu is currently a full-time PhD student in both NICTA and the University of New South Wales (UNSW). His current research topic is "Recovery for Operations on Cloud Applications". Prior to starting his PhD study, he has worked in several Singapore IT industry companies for more than 5 years as software engineer, system analyst, analyst developer, etc.



Liming Zhu is the Research Director of Software and Computational Systems in Data61, CSIRO, Australia. He holds conjoint positions at University of New South Wales (UNSW) and University of Sydney. He is a committee member of the Standards Australia IT -015 (system and software engineering) contributing to ISO/SC7. His research interests include software architecture and dependable systems.

Guoqiang Li received his BS degree, MS degree, Ph.D. degree from Taiyuan University of Technology in 2001, Shanghai Jiao Tong University in 2005, and Japan Advanced Institute of Science and Technology in 2008, respectively. He worked as a postdoctoral research fellow in Graduate School of Information Science, Nagoya University, Japan during 2008-2009, and as an assistant professor in School of Software, Shanghai Jiao Tong University, during 2009-2013. He now is an associate professor, School of Software, Shanghai Jiao Tong University, China.

Qinghua Lu is a lecturer at Department of Software Engineering, China University of Petroleum, Qingdao, China. She received her PhD from University of New South Wales (UNSW) in 2013. Her research interests include dependability of cloud computing, architecture of big data applications, and service engineering.