2017 IEEE 14th International Conference on Services Computing

Lightweight and Adaptive Service API Performance Monitoring in Highly Dynamic Cloud Environment

Jingmin Xu *†, Yuan Wang †, Pengfei Chen †, Ping Wang *
* School of Software and Microelectronics, Peking University,
† IBM Research China
{xujingm, crlwangy, cpfchen}@cn.ibm.com, pwang@pku.edu.cn

***Abstract*—Cloud platforms and services usually provide an API layer as decoupled, language agnostic interface for both front-end client integration and back-end data and/or function access. The availability and performance of the APIs have significant impact on the quality of end user or client experiences due to its nature of interaction endpoints. However, the extreme dynamics, complexity and scale of the current cloud platforms challenge the applicability of the existing performance monitoring and anomaly detection approaches from timeliness, accuracy, and scalability perspectives. This paper presents a novel approach to API performance monitoring, which recognizes performance problems by response time deviation from a baseline response time / throughput model that are created and continuously updated through online learning. In the post-detection phase, an MIC (Maximal Information Criteria) based correlation algorithm is used to group alerts into a higher level and more informative hyper-alerts for end user notification. We prototyped our solution for a large-scale commercial cloud platform, evaluated it using three months' API performance metrics data, and compared with a couple of existing representative algorithms and tools. The results show our approach is able to detect API performance anomalies with a high F1-score. Compared to existing Granger based approach, our approach has achieved nearly one time increase in F1-score. Moreover, the alert reduction ratio of our approach outperforms several state-of-the-art approaches.**

***Index Terms*—Service; API; Anomaly detection; Alert reduction;**

I. Introduction

The popular cloud platforms and services from Amazon[1], IBM[2], Salesforces[3], etc. provide an API layer as decoupled, language agnostic interface for both front-end client integration and back-end function access. The availability and performance of the APIs have significant impact on the quality of end user or client experience due to its interaction endpoint nature. The API gateway that manages API lifecycle is usually designed to provide the lowest possible latency for API requests/responses at large scale as well as monitor and throttle traffics that may overwhelm the back-end services at rush hours. One of the challenges for API gateway is to detect API performance anomaly timely and accurately so as to proactively manage traffic in a just-in-time fashion for the best user or client experience and satisfaction. There have already been many studies in the area, but the applicability of the existing algorithms and tools [4] to API monitoring in highly dynamic, complex and large-scale cloud environments faces challenges from timeliness, accuracy and scalability perspectives.

The anomaly detection approaches most deployed in today's data center apply a threshold on the metrics being monitored. Often these thresholds are applied to each individual measurement separately, e.g. response time is more than 10 seconds or CPU utilization is over 80%. Variants such as Multivariate Adaptive Statistical Filtering (MASF)[5] additionally maintain a separate threshold for data segmented and aggregated by time (e.g., hour of day, day of week), thus cater for seasonal variation in workload, e.g., heavier load during the week and lighter load over the weekend by using different thresholds for different seasons. However, there are some serious issues with these approaches. First, selecting an optimal threshold is a pretty difficult task, especially for people who don't have much knowledge about the system being monitored. They usually set it too high or too low and have to keep adjusting it to the level they are comfortable with. Second, static or periodic based dynamic thresholds cannot adapt to loads that may change over time or intermittent bursts, nor can they react to anomalous behavior that may not show up as extremal large or small values in the data[6]. The last is about one more fundamental issue that abnormal states are difficult to describe with simple metrics and associated thresholds. Detecting performance anomalies by predefined rules with combined use of multiple metrics does not always work as expected. One system that exhibits very good responsiveness to user requests as being with more than 80% of server CPU utilization may instead have a seriously high latency when server CPU utilization goes down to 20%. Apdex[7] converts many measurements into one number on a uniform scale of 0-to-1 (0 = no users satisfied, 1 = all users satisfied). The resulting Apdex score is a numerical measure of user satisfaction with the performance of enterprise applications. However, Apdex is more suitable for being a high level reporting mechanism that shows overall application trends via an index, instead of being used as a performance indicator for real time anomaly detection [8]. All of these issues lead to increased false alarms and reduced accuracy with these approaches.

A number of prior anomaly detection algorithms, which take either reactive approaches or a proactive approaches, use various modeling techniques that mandate a certain level of understanding of the system structure or behavior to build a model, or learning from a potentially large amount of historical data to discover a model. It can be a system performance model used to recognize performance deviation[9-12], a metric correlation model used to identify performance problems that break the correlation[13-17], or a prediction model used to predict when the problem will happen and on which machine[23-26]. However, the extreme dynamics of today's cloud platforms constantly break the established models due to the following operations: (1) continuous update or upgrade of infrastructure components; (2) dynamic resource allocation and consolidation for workload optimization; (3) horizontal or vertical resource scaling for elastic computing; (4) continuous deployment or update of applications or services with diverse workload characteristics; (5) co-existence of multiple versions of applications or services for A/B testing. Keeping the models current by manual updates or offline re-training incurs potentially significant overhead especially when the monitoring granularity is eventually refined to the individual API level as it

2474-2473/17 $31.00 © 2017 IEEE
DOI 10.1109/SCC.2017.80

is usually tens of thousands of APIs to monitor and manage in a large-scale cloud platform.

Based on the above discussions, an API performance monitoring approach should have the following characteristics in order to be applied to high dynamic and large-scale cloud environments: (1) threshold-less: no need to specify any threshold; (2) adaptive: be able to continuously adapt itself to constantly changing backend environments; (3) lightweight / efficient: both in terms of the number of metrics required to run (the volume of monitoring data continuously captured and used), and in terms of their runtime complexity for executing the detection methods; (4) independent: no needs to deploy sensors on the backend system or infrastructure, and even no need to integrate with the existing backend monitoring system; (5) high true positive rates and low false positive rates.

In [27], the authors present a target-less and model-free approach for a self-optimizing application workload manager, which takes application as a black box and use admission control to keep the system running at the best operating point where the measure Power, which is average throughput divided by average response time, is maximum in a control cycle. It assumes that the best performance can be described by the maximum value of Power for any given throughput, and any meaningful deviations, e.g. more than 10%, from the best can be regarded as congestions. However, this is a very strong assumption given the complex and dynamic relationship between the two metrics, which leads to increased false alarms and reduced accuracy with the approach.

The approach we propose in the paper also takes the backend platform and services as a black box because it brings two benefits: (1) lightweight due to significantly reduced numbers of metrics with a focus on response time and throughput; (2) independent without having to integrate with back-end monitoring system. But it takes a different way of detecting performance anomalies based a couple of assumptions that are intuitive to understand and being validated by many of our observations: (1) in an stable environment, the response time distribution under any specified throughput is Gaussian distribution which has been verified with our real production data, where the values for both mean and standard deviation are different from throughput to throughput; (2) performance stability for a time can be measured by the standard deviation of the response times over the time; (3) performance anomalies can be detected on data points (response time) falling outside of the normal range under the same throughput while the performance stability keeps going down for some time. As for (1), instead of mining an immense amount of historical data, we employ an online machine learning based approach to create and incrementally update baseline models about response time against throughput, which makes this approach much more adaptive to the backend platform and services changes.

Getting to API level monitoring usually brings the benefits of alerting accuracy because exact models and thresholds can be set up for different APIs. On the other hand, it may exacerbate the problem of alert storm because the same performance anomalies may be alerted repeatedly by multiple related or dependent APIs. In order to mitigate this issue, we employ an MIC based correlation algorithm to automatically group raw alerts into higher level, more informative hyper-alerts for end user notification.

We prototyped our solution for a large scale commercial cloud platform and evaluated it using three months' API performance metrics data and labelled anomalies against a couple of existing representative algorithms and tools. The results shows our solution is able to detect API performance anomalies with lower false positive/negative rates and longer lead time in such a highly dynamic cloud environment.

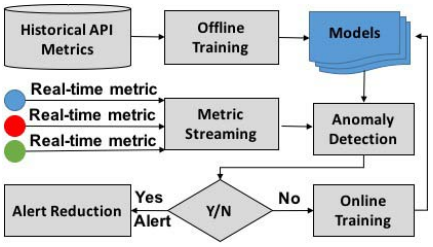

Fig. 1. The overview of our system

The remaining of the paper is organized as follows. Section II presents an overview of our approach. We will introduces the design details and related algorithms in Section III. Section IV presents the evaluation results of our prototype against a real-world cloud platform. Section V presents the related works. Section VI concludes the paper and presents the future work.

II. Overview

This section presents the high level components and process of our solution. As you can see in the Figure 1, there are basically five components involved in the process.

- **Offline Training**: it is used to train the baseline response time / throughput models against historical data sets if any. The trained models can be used as input to warm start the Anomaly Detection component. However, this component is optional because Anomaly Detection component can be cold started without any models as input.
- **Metric Streaming**: it is used to capture and collect performance metrics, i.e. API response time for each request, and create a metric stream so that data can be processed sequentially and incrementally.
- **Anomaly Detection**: it is used to detect performance anomaly against the baseline models. The component can be either warm or cold started. It directly passes on the metric stream to the Online Training component for online model training if the baseline models are not available. Once an anomaly is detected, an alert will be fired with the anomaly information.
- **Online Training**: it is used to train the baseline models if they are not available, or tune the models with the latest received metrics if no performance anomaly is detected.
- **Alert Reduction**: it is used to group the raw alerts into reduced number of higher level, more informative hyper alerts for end user notification.

III. API Performance Anomaly Detection

In this section, we present the design and algorithms of our performance anomaly detection approach in details.

A. Assumption

As described in Section I, the approach takes the backend platform and services as a pure black box and doesn't assume any knowledge about the dynamic resource consumption of the backend. It continuously captures the only performance metric - response times of APIs and uses it together with other derived metrics in the anomaly detection algorithm. Instead of simply

deriving baselines and thresholds based on an assumed metric distribution model [5][29], the approach takes into account the two observations that have being validated by many of our practices: (1) the response time distribution at a time is majorly determined by the throughput of the time. The Gaussian distribution is the assumed underlying probability model, but the mean and standard deviation for the distribution are different from throughput to throughput. The models that reflect the relationship between response time (for both mean and standard deviation) and throughput are created and continuously updated by on-line machine learning for being adaptive to backend dynamics. (2) the data points falling outside the normal range of the distribution are recognized as anomalies only if the performance stability keeps going down for some time. The performance stability for a time is measured by the standard deviation of response times over the time. The smaller the value is, the more stable the performance is. When the performance stability goes up, the execution of the API is entering into a more steady state, and vice versa. Firing anomalies only if the execution of APIs is leaving further away from the steady states is able to ameliorate false alerting against the bursts of response times.

B. Metrics and Baseline Models

Table 1 shows all the metrics used in the anomaly detection algorithm.

$BLResT_{\theta 1}(Thr)$ and $BL\sigma_{\theta 2}(Thr)$ are the models between response time (for both $AvgResT$ and $ResT\sigma$) and throughput, which are used as the baseline for identifying response time deviations as potential anomalies. The baseline is created and continuously updated with the latest metric data through on-line machine learning. The features chosen for polynomial regression can be expressed as $X = (Thr^1, Thr^2, Thr^3, \cdots)^T$.

Our experiment shows Thr^7 is the upper bound of the maximum order feature in order to balance the high bias and overfitting of the machine learning algorithm. $BLResT_{\theta 1}(Thr)$ and $BL\sigma_{\theta 2}(Thr)$ can be computed by the following hypothesis functions:

$$BLResT_{\theta 1}(Thr) = \theta 1^T X, BL\sigma_{\theta 2}(Thr) = \theta 2^T X$$

In order to bootstrap the learning, the initial hypothesis functions are prepared for two different scenarios: (1) if sufficient amount of historic data is available, they can be used to run a regression as the initial hypothesis functions; (2) if no historic data is available, $\theta 1$ and $\theta 2$ can be set to:

$$\theta 1^T = (\theta 1_0, 0, 0, \cdots, 0), \theta 2^T = (\theta 2_0, 0, 0, \cdots, 0)$$

$\theta 1_0$ is the estimated response time and $\theta 2_0$ is the estimated standard deviation of the response time. The estimation accuracy doesn't matter because $\theta 1$ and $\theta 2$ will be eventually converged through online learning with streaming metric data.

C. Anomaly Detection Algorithm

The algorithm to detect API performance anomaly is shown in Algorithm 1. The algorithm recognizes the potential anomalies by response time deviation from the baseline models:

$$AvgResT \geq BLResT_{\theta 1}(Thr) + 3 * BL\sigma_{\theta 2}(Thr).$$

Once the potential anomalies are identified, the algorithm decides whether any alert needs to be fired by examining $ResT\sigma(n)_{recent}$ to see if the performance stability keeps going down for some time. If no alerts is eventually fired and the performance stability keeps going up, the baseline model needs to be updated with the latest metrics by an on line learning algorithm which has O(n) time complexity, and n is the number of the APIs. In this algorithm, there are some pre-set parameters such as learning rate and stable_factor. These parameters are set by trying and comparing among some experiment values on the historic data to maximum the performance of the algorithm.

Algorithm 1 API Anomaly Detection

Input: All request's $ResT$ in current time window, $TWCount$, $ResT\sigma(n)_{recent}$, $BLResT_{\theta 1}(Thr)$, $BL\sigma_{\theta 2}(Thr)$, and learning rate α.
Output: raw_alert, updated $ResT\sigma(n)_{recent}$, updated $BLResT_{\theta 1}(Thr)$, $BL\sigma_{\theta 2}(Thr)$.

Let $AvgResT = 0$, $Thr = 0$, $sum = 0$, $stable\_factor = 0.8$
for all request in TW **do**
  compute $AvgResT$
  count Thr
  compute $ResT\sigma$
end for
$raw\_alert = false, unstable\_count = 0$
if $AvgResT - BLResT_{\theta 1}(Thr) > 3 * BL\sigma_{\theta 2}(Thr)$ **then**
  for $i = 1$ until $(TWCount - 1)$ **do**
    if $ResT\sigma(i+1) > ResT\sigma(i)$ **then**
      $unstable\_count \leftarrow unstable\_count + 1$
    end if
    $i \leftarrow i + 1$
  end for
  $unstable\_percentile = unstable\_count/(TWCount - 1)$
  if $unstable\_percentile \geq stable\_factor$ **then**
    $raw\_alert = true$
  end if
else if $(raw\_alert = false)$ and $(1 - unstable\_percentile) \geq stable\_factor$ **then**
  online learning by a new sample point $(Thr, AvgResT)$ and $(Thr, ResT\sigma)$, update the regression parameters $\theta 1$ in $BLResT_{\theta 1}(Thr)$, and $\theta 2$ in $BL\sigma_{\theta 2}(Thr)$ in the following way:
  $\theta 1_j \leftarrow \theta 1_j - \alpha * (BLResT_{\theta 1}(Thr) - AvgResT) * Thr$
  $\theta 2_j \leftarrow \theta 2_j - \alpha * (BL\sigma_{\theta 2}(Thr) - RestT\sigma) * Thr$
end if
remove $ResT\sigma(1)_{recent}$ value
change $ResT\sigma(n-1)_{recent}$ with $ResT\sigma(n)_{recent}$, where $n = 2, 3, 4, \cdots, TWCount + 1$
$ResT\sigma(TWCount)_{recent} \leftarrow ResT\sigma$

IV. Alert reduction

In the above section, we have demonstrated the details of anomaly detection procedure on API metrics. However, in real-world large-scale commercial production systems, there can be tens of thousands of APIs. Hundreds of alerts are generated by the monitor system simultaneously especially when some back-end services (e.g, Database services) are broken. In this section, we will demonstrate a post-mortem approach to correlate and reduce alerts.

TABLE I
METRICS NOTATION

Notation	Definition
TW	Time window length for collecting and computing the latest performance metrics
$TWCount$	Number of the time windows TW for accumulating and computing the recent metrics
$ResT$	Response time of a specific API request
$AvgResT$	Average response time of a specific API request within a time window TW
Thr	Throughput of a specific API request with in a time window TW
$ResT\sigma(n)$	The standard deviation of response time of an API in a time window, which reflects the performance stability of the API
$ResT\sigma(n)_{recent}$	The list of the standard deviation of response time in the past n time windows, the maximum value of n is $TWCount$
$BLResT_{\theta 1}(Thr)$	The model for the relationship between $AvgResT$ and Thr, which is a polynomial regression function with parameter vector $\theta 1$
$BL\sigma_{\theta 2}(Thr)$	The model for the relationship between $ResT\sigma$ and Thr, which is a polynomial regression function with parameter vector $\theta 2$

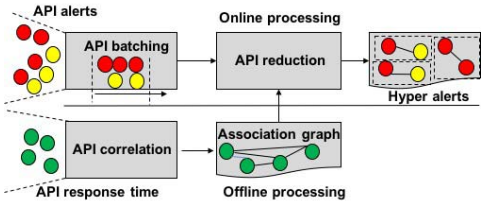

Fig. 2. The framework of alert reduction

The rationale behind the alert reduction is that some alerts are highly correlated as there are logically dependent relationships amongst these services. For example, a front-end presentation API depends on the business logic processing API hosted in the middle tier. The correlated alerts can be combined as a single alert which is called "Hyper alert" in this paper. However, solely from the API level, we cannot directly know the logical dependencies amongst these APIs. A common practice is to discover these dependencies by correlating the response time or throughput of APIs. The merit of this approach is that the dependent relationships amongst APIs can be inferred in a black-box way. Based on this rationale, we propose our alert reduction approach shown in Figure 2. Our approach mainly contains two parts, namely an offline processing part and an online processing part. In the offline part, we correlate different APIs by checking the correlation relationship of response times of these APIs. The output is an association graph composed by APIs. The online part mainly consists of two modules, namely alert batching and alert reduction. The alert batching module is in charge of separating the arriving alerts into batches. Each batch contains a bunch of alerts. The batch of alerts are then fed into alert reduction module to group these alerts into "Hyper alerts". The association graph generated in the offline part is an input in the alert grouping procedure. With the "Hyper alerts", system operators only need to investigate very few alerts to find the root causes. Moreover, it can help the operators understand the alert propagation path.

A. Alert batching

When conducting alert reduction, we should know in advance which alerts should be reduced. Alert batching aims to resolve this problem. Commonly, if two API alerts have direct "cause-effect" relationship, the delay between these two alerts should stay in a short-time window which is denoted as ω and set as 2 minutes in this paper [1]. If no new alerts arrive during such a time, the previous alerts are packed in a batch and fed into the alert reduction module. For example, $API_1^a(t_1)$, $API_2^a(t_2)$, $API_3^a(t_3)$, $API_1^a(t_4)$, and $API_4^a(t_5)$ represent a series of alerts on different APIs and sequentially arrive at t_1, t_2, t_3, t_4, and t_5 respectively. We have $t_2 - t_1 < \omega$, $t_3 - t_2 < \omega$, $t_4 - t_3 < \omega$, and $t_5 - t_4 > \omega$. Hence, API_1^a, API_2^a, API_3^a, API_1^a form a batch. It is worth noting that there may be multiple API alerts in the same time slot.

[1] ω may need to be adjusted in different systems.

B. Alert correlation

As for alert correlation, there are multiple approaches proposed from different perspectives. One intuitive approach is to correlate different alerts via scope. A scope denotes the location or component that a alert comes from. The scope information is always one part of the alert. For instance, from the alert name "node1.alert.cpu_high", we know this alert is generated in node1. Thus, the alerts with the same scope can be correlated together. The scope-based approach is commonly used in alert management tools (e.g., IBM Netcool). However, this approach cannot correlate alerts across scopes. Besides, statistical correlation based approaches are also commonly adopted. In prior arts [30], Pearson correlation coefficient is the first solution. While this statistical approach cannot discover the non-linear or non-functional relationships amongst these alerts. From the temporal perspective, Granger causality [31] is an efficient way to discover the causality between any two alerts. However, Granger causality assumes the lag between two alerts are stable [32]. This assumption does not always hold in real-world API ecosystems due to high dynamics.

To capture the correlation between a pair of API metrics, a state-of-the-art method named Maximal Information Criteria (MIC) [33] is introduced. MIC has a great power to capture the complex correlation relationships. To keep the paper self-contained, we give several definitions and preliminaries about MIC first.

Definition 1. *Resolution Grid: Given a finite set D of ordered pairs, the x-values of D is partitioned into x bins and the y-values of D is partitioned into y bins, allowing empty bins. Such a pair of partitions an $x - by - y$ resolution grid.*

Given a grid G, let $D|_G$ be the distribution induced by the points in D on the cells of G. For a fixed D, different grids G result in different distributions $D|_G$.

Definition 2. *Mutual Information: The mutual information between two random variables X and Y is defined as:*

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) log(\frac{p(x,y)}{p(x)p(y)})$$

where $p(x,y)$ is the joint probability distribution function, $p(x)$ and $p(y)$ are the marginal probability distribution function.

Definition 3. *Highest Mutual Information: For a finite set $D \subset R^2$ and positive integers x, y, define*

$$I^*(D,x,y) = maxI(D|_G)$$

where the maximum is over all grids G with x columns and y rows, and $I(D|_G)$ denotes the mutual information of $D|_G$.

Definition 4. *Characteristic Matrix: The characteristic matrix $M(D)$ of a set D of two-variable data is an infinite matrix with entries:*

$$M(D)_{x,y} = \frac{I^*(D,x,y)}{log \quad min\{x,y\}}$$

Definition 5. *Maximal information coefficient (MIC):The Maximal Information Coefficient (MIC) of a set D of two-variable data with sample size n and grid size less than $B(n)$ is given by:*

$$MIC(D) = max_{xy<B(n)}\{M(D)_{x,y}\}$$

where $\omega(1) < B(n) \leq O(n^{1-\varepsilon})$ for some $0 < \varepsilon < 1$

Just as pointed in [33], we also use $B(n) = n^{0.6}$. The detailed description of the definitions and the boundary of some variables could be found in the supporting online material of [33]. The procedure to calculate MIC includes the following steps:

- **step 1**: Find the approximative highest mutual information for data D, namely $I^*(D,x,y)$. The goal of this step is to find a optimal $x-axis$ partition given fixed $y-axis$ partition using dynamic programming. Refer to the supporting online material of [33] for details.
- **step 2**: Construct the characteristic matrix using the obtained $I^*(D,x,y)$ according to the definition.
- **step 3**: Calculate the MIC for D by looking for the maximum value in the characteristic matrix.

It is always hard to give the complexity of MIC algorithm precisely. According to the supporting online material of [33], the time complexity of MIC could be roughly defined as $O(\hat{k}^2xy)$, where is a preset parameter (i.e., a constant) and x and y are the partition on data set D mentioned in the description of MIC. Therefore, even for a large data set, it is still affordable for MIC.

For each API pair (X,Y), their association coefficient is represented by the $MIC(X,Y)$ score which falls in the region $[0,1]$. In this paper, a simple but exhaustive pair-wise method is adopted to calculate all the associations. Suppose M metrics are collected, $M(M-1)/2$ association pairs should be obtained. However, some association scores are not significant enough to determine the existence of associative relationship. In this paper, we provide a threshold $\epsilon = 0.05$ to filter the associative pairs. Once $MIC(X,Y) > \epsilon$, the associative relationship exists. In previous work [30], Pearson correlation coefficient is commonly used to judge the correlation between two metrics. To compare the effectiveness of Pearson correlation coefficient and MIC in discovering the associative relationships, we select 50 APIs and determine the associative relationships amongst the response times of these APIs with these two methods. The results are shown in Figure 3 and Figure 4 respectively. The number on X-axis and Y-axis denotes the ID of each API and the yellow dot points represent associative relationships between any two APIs. From these two figures, we can observe clearly that the our MIC-based approach can discover more associative relationships than

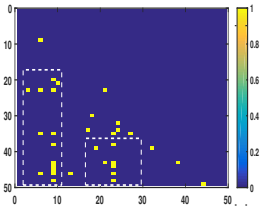

Fig. 3. Result obtained by MIC

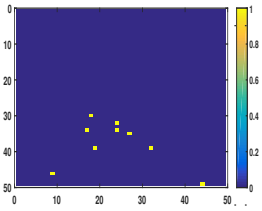

Fig. 4. Result obtained by Pearson

Pearson correlation coefficients-based approach. After investigating the associative relationships which are missed by Pearson correlation coefficients, we find that these relationships do not show direct linear functional relations. But in reality, these APIs indeed have dependent relations. The excellence of MIC-based approach can help increase the effectiveness of alert reduction which will be seen in the experimental section.

C. Alert reduction

When a new batch of alerts arrive, the art reduction module will separate these alerts into different groups. Every alert in each group is correlated with one or multiple other alters. In the other words, the alerts in each group are connected in the alert association graph. Moreover, these grouped alerts which are named as "hyper alert" can provide contextual information to help operators understand why and how these alerts are generated. To group the given alerts together, we need to traverse the alert association graph to find out the connected subgraph. Given a batch of alerts, $\mathbf{A} = (API_1^a(t_1), API_2^a(t_1), API_2^a(t_2), API_3^a(t_3), \cdots, API_k^a(t_i))$, we first combine the same kind of alerts which are generated at different times into one alert. After the combination, we get a new batch of alerts, namely $\bar{\mathbf{A}} = (API_1^a, API_2^a, \cdots, given alters such as API_1^a. From the alert association graph, we find out the maximal subgraph that contains the selected alert with a Depth-First-Search (DFS) traversal approach. Meantime, every alert in this subgraph should be contained in $\bar{\mathbf{A}}$. This subgraph is recognized as a hyper alert. Then we exclude the alerts in the subgraph from $\bar{\mathbf{A}}$. From the remaining alerts, we randomly select another alert and repeat the above steps until there are no remaining alerts. Finally, we obtain all the hyper alerts for this batch of alerts. To clarify the procedure of alert reduction, the pseudocode is shown in Algorithm 2. The time complexity of this algorithm is dominated by the DFS traversal procedure. It can be calculated as $O(n^2)$ where n denotes the number of edges in the association graph.

V. Experiment

In this section, we perform comprehensive experimental evaluations for the proposed anomaly detection and reduction methods. We evaluate the effectiveness and performance of our algorithm on real commercial cloud platform with metrics, Precision, Recall, F1-score, and alert reduction ratio. The alert reduction ratio denotes the ratio of grouped hyper alerts in the raw alerts. F1-score is defined as:

$$F1 - score = \frac{2 * Recall * Precision}{Recall + Precision}$$

Algorithm 2 Alert reduction algorithm

Input: $\mathbf{A} = (API_1^a(t_1), API_2^a(t_1), \cdots, API_k^a(t_i))$ is a batch of alerts.

Output: A set of hyper alerts, $hyper\_alert$

1: $\bar{\mathbf{A}} \leftarrow Combine(\mathbf{A})$ // Combine the the same kind of alerts at different times.
2: **While** $\bar{\mathbf{A}}$!= NULL
3: $r\_alert \leftarrow RandomSelect(\bar{\mathbf{A}})$; // Randomly select one alert.
4: $hyper\_alert[i] \leftarrow FindSubgraph(r\_alert)$ // Find the maximal connected subgraph containing r_alert.
5: $\bar{\mathbf{A}} = \bar{\mathbf{A}} \setminus hyper\_alert[i]$

A. Experimental setting

We use a real large-scale commercial public cloud as our testbed. This cloud provides both of Infrastructure as a service (IaaS) and platform as a service (PaaS) through a single set of backend APIs. It is deployed and runs across multiple data centers all over the world. In each data center, it runs on thousands of machines. There are hundreds of micro-services cooperating with each other to provide services to end users. To keep this huge platform running continuously and reliably, there are a dedicated SRE (Site Reliability Engineer) team intensively watching and fixing anomalies. There are two types of APIs in this cloud, namely platform APIs and user application APIs, which are the APIs to control and manage the platform itself and the APIs provided by the application running on this cloud respectively. In a single data center, the number of the first type of APIs is about 3000+ and number of the second type of APIs is about 8000+. Due to resource share and the frequent version updates, bug fix, auto-scale, auto-recovery operations, these APIs are all running in a highly dynamic environment. Actually, to monitor these APIs, there are about 30 million metric points collected each hour, so with this metrics ocean, how to help SRE effectively monitor 11000+ APIs' performance in the same time and generate meaningful alters is a big challenge.

In our experiment, we collect metric data for three months, and prove the scalability of our method by handling all 11000+ APIs' monitoring and performance anomaly detection using a small Spark streaming cluster with 64 CPU cores and 128GB memory as our computing resources and a Kafka cluster with 16 CPU cores and 32GB memory as the metric pipeline cache. To validate the effectiveness of algorithm, we need known anomalies for these APIs, since it is not practical for cloud SRE to label all 11000+ APIs' performance anomalies manually. Therefore, we choose several APIs which SREs are most familiar with and are most important to cloud end user, namely APIs of platform Web Console and Command Line Tool. To validate the API anomaly grouping method, we considering all the 3000+ platform related APIs since the grouping result is easy for SRE to validate.

B. Anomaly detection result

Anomaly detection. Figure 5 to Figure 8 show an example of detected anomaly and baseline update, these figures are plotted based on a selected segment of real data for a specific API. Figure 5 shows the variation of $BLResT_{\theta 1}(Thr)$ which is a polynomial regression function to describe the baseline of response time under a throughput. This curve may change over time by an on line learning algorithm. The red curve is the baseline at time t_i and the blue curve is the updated baseline at time t_k. Figure 6 shows the $BL\sigma_{\theta 2}(Thr)$ which describes the standard deviation of the baseline of response time under a throughput. The red curve is the baseline at time t_i and the blue curve is the updated baseline at time t_k. Figure 8 shows the $ResT\sigma(n)_{recent}$, which is the standard deviation of response time in each time window. The red curves in Figure 5, Figure 6 and Figure 7 together show an anomaly detection scenario, namely the red start point in Figure 5 denotes the response time is above the region of the response time baseline under the same throughput. Here the value of σ is determined by the point under the same throughput in the red line of Figure 6. But this is not a sufficient condition, we still need to examine the $ResT\sigma(n)_{recent}$. In Figure 7, we can see before t_i, there are more than 30 time windows within which the response time's standard deviation keeps going up meaning that the API's performance becomes more and more unstable. Later, an alert is fired.

Baseline update. As described in Section III, the baseline needs to be updated if the API runs into a new stable status. The Blue curves in Figure 5, Figure 6 and Figure 8 together show a baseline update scenario, namely the blue start point in Figure 5 denotes that at current time t_k, the response time is smaller than the response time baseline under the same throughput. We need to examine the $ResT\sigma(n)_{recent}$ to see if the baseline needs to be updated. In Figure 8, we can see that before t_k, there are more than 30 time windows within which the response time's standard deviation keep going down, this means the API's performance becomes more and more stable, so the baseline in Figure 5 and Figure 6 will be updated. Now the blue star point in Figure 6 will be used to do online learning to update the regression parameters, and the new baseline is shown as blue cure in Figure 6. Actually, after talking with SRE, we confirmed these updates were caused by the two instances that were added into the back-end cluster for addressing a throughput related issue.

Evaluation for anomaly detection. The proposed anomaly detection method is compared with other three state-of-the-art methods, namely static threshold, dynamic threshold with periodic workload, and Granger causality based prediction methods. The number of the known performance anomalies is 261 in total in three months. All these anomalies are significant anomalies which impact end users. In the public cloud monitoring system which is used in this validation, the anomaly reported by our method discussed in this paper is defined as "raw alert". Based on these raw alert, SRE will define some so called "situations" o report real anomalies. The situation is usually defined as a complex event evaluation expression, such as "there are more than 30 raw alerts are fired again an API within 10 minutes and the reported recovery number is smaller than 15% out of the raw alerts". So in order to be fair in comparing among different methods, we use the same situation definition for all the above methods. For the static threshold based method, we set 30 seconds which is the real threshold in production as the threshold for these APIs. For the dynamic threshold with periodic workload, we assume the workload period is weekly and the time range to compute the average value of response time is half an hour. For the granger causality based prediction, we use 2 weeks' data before the evaluation to train the model. For the method proposed in this paper, we set the $TWCount$ to 60 which can reflect the API performance during recent 10 minutes. And we set on line learning rate $\alpha = 0.01$. The validation result is shown in Table 2 and Figure 9. Table 2 shows the

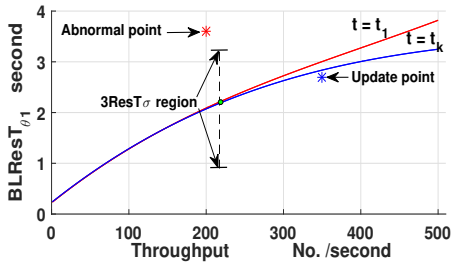

Fig. 5. The fitting curve between API throughput and response time

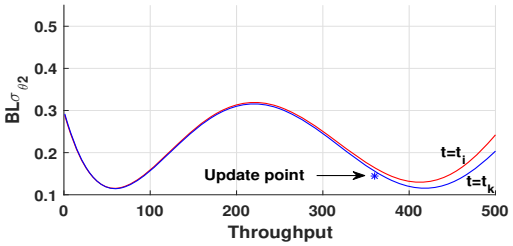

Fig. 6. The fitting curve between API throughput and σ of response time

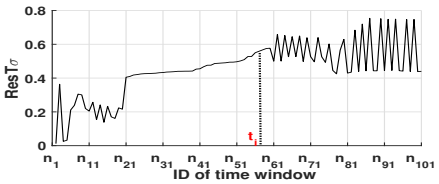

Fig. 7. The increasing scenario of σ along with time

number of true positive, false positive, false negative and the detection lead time of different methods. From the results, we can see that our method can detect most (249) of the labelled anomalies (261), and report the second small number of false alert (35). Dynamic threshold report the 182 anomalies out of 261 in total, but sends out huge number (1210) of false alerts. This is caused by two reasons. One is the periodic workload assumption may not hold under this dynamic environment. The other one is the API itself and its running environment may change during three months. The Granger causality based method reports the majority of the labelled anomalies but fires many false alerts. This could be caused by the statistic model changes during three months. Actually, there are 2 major version updates of this API during this three months. Therefore, each time after the updates, this method's performance goes down significantly. The static threshold method achieves a very small number of false alerts (17) because 30 seconds is a very conservative value for our target APIs. But this conservative value can also cause it missing some anomalies(58). Another reason that the static threshold has the second best result is that the SREs are very familiar with this API's characteristics and set a good threshold based on their experience. But for all 11000+ APIs, it is impossible for SREs to set proper threshold one by one. For the lead time comparison, we use the time which static threshold report the anomaly as the base which is also the time when the real production system receives the anomaly. We can see that our method is 11.6 minutes ahead, the static threshold method is 0 minute, the Granger causality method is 5.9 minutes ahead, and dynamic threshold is 2.2 minutes ahead. This could be explained by the following reasons. Our method detects "trends" of the performance and ires anomaly before the response time exceeds 20 seconds. The Granger causality detects the changes of the causality relationship between response time and throughput. Therefore, it could find anomalies before the API's performance goes too bad. For the dynamic threshold based method, the long response time value needs to be accumulated to bigger than the historical average value of the same time slot. However, it also does not need to wait to exceed 30 seconds. Overall, our method has the best lead time and this will give SREs important chances to take more proactive actions to prevent more serious incidents.

The Figure 9 shows the precision, recall and F1-score for each compared method. The result shows that the proposed method holds the second place in precision, the first place in recall and F1-score. We can conclude that our methods achieves the best results. The static threshold based approach is not as good as ours, but it is a feasible option because it can be elaborately adjusted by experienced SREs. The other two approaches have

TABLE II
TP, FP, FN AND LEAD TIME COMPARISON OF DIFFERENT APPROACHES

Measurement	Proposed method	Static	Dynamic	Granger
TP	249	203	182	231
FP	35	17	1210	526
FN	12	58	89	30
$LeadTime$	-11.6m	0m	-2.2m	-5.9m

an obvious gap and still need a lot of tuning and customization for achieving the same level of results.

C. Alert reduction

The above session has shown that our anomaly detection method can detect system anomaly accurately. However, some of alerts can be grouped together due to high correlations. This section shows the effectiveness of our alert reduction approach. In the evaluation experiments of anomaly detection, we adopt 261 confirmed significant anomalies to validate the effectiveness. However, for alert reduction, these alerts are not enough. Therefore, we tune the threshold of our anomaly detection approach to generate more alerts. In this experiment, we generate more than 3600 alerts with one month's API metric data. Moreover, these alerts are put in 39 consecutive batches. Figure 10 shows the numbers of raw alerts and the grouped hyper alerts. From this figure, we can observe that the raw alerts are reduced significantly. In batch 25, the number of raw alerts is reduced from 110 to 19. For the system operators, it is not necessary to analyze the raw alerts one by one. They only need to focus on the grouped hyper

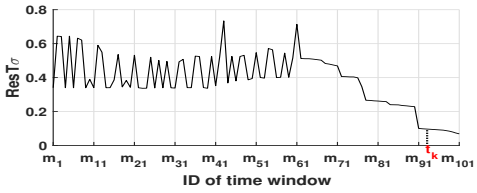

Fig. 8. The decreasing scenario of σ along with time

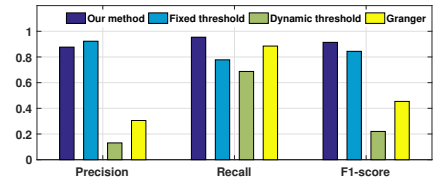

Fig. 9. The anomaly detection results of different approaches.

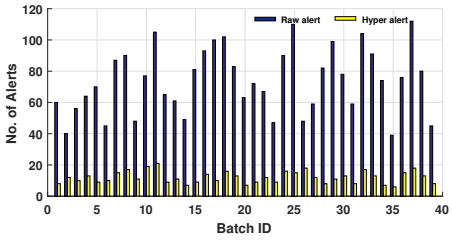

Fig. 10. The result of alert reduction

alerts in order to investigate system problems. To further validate the effectiveness of our alert reduction approach, we compare the alert reduction result of our approach with Pearson correlation coefficient and the Granger causality based approaches. Figure 11 shows the max, min, and average alert reduction ratios of different approaches. This figure demonstrates that our approach outperforms the other two approaches no matter in max, min, or average reduction ratio. The advantage of our approach is attributed to the powerful association discovery capability of MIC. MIC can discover more associative relations than the other two approaches. Therefore, the alert reduction ratio is high.

VI. Related work

As discussed in the introduction part, the most relevant work is [27]. We were inspired by its black box, target-less and model-free based approach, but its assumption of having the single, best operating point manifested by the maximum value of the measure Power is too strong. Keeping the system running at the best operating point by request admission seems feasible for workload management, but deviations from the best may not necessarily indicate any performance anomalies. Instead of purely basing on heuristic algorithms, our approach recognizes performance anomaly by deviations from a baseline model that is online learnt and continually updated so as to be more adaptive to workload and infrastructure changes.

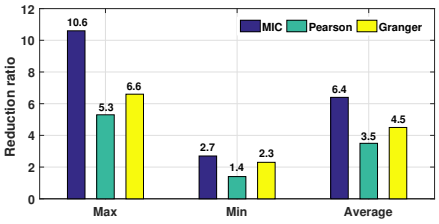

Fig. 11. The comparison of different alert reduction pproaches.

There are model based approaches represented by [25] and [26] that are designed to inherently accommodate to changes. ALERT[25] is an adaptive runtime anomaly prediction system that can raise advance alert before an anomaly happens. During runtime, ALERT is able to dynamically switch between different prediction models based on context evolving patterns to achieve high quality anomaly prediction for dynamic systems. The explicit mapping from prediction models to different execution contexts makes the models context aware and avoids repetitive model training. However, it depends on an anomaly detection system to provide normal and anomaly state labels for different measurement samples. Cherkasova et al. [26] proposes an integrated framework of measurement and system modeling techniques to detect application performance changes and differentiating performance anomalies and workload changes. Therefore, it is able to avoid false alarms raised by the algorithm due to workload changes. Both of them claims to be black-box based approach, but they need to deploy monitoring sensors on all nodes in the hosting infrastructure, which continuously monitor a set of resource consumption or performance metrics for each running node and application components. This makes them less independent, adaptive and efficient for large-scale, highly dynamic cloud platforms and services.

Instead of detecting violations on metric values, there are some existing work [18-22] that use information and probability theories to detect anomalies in metric distribution. Among others, EbAT[18-20] aims to address the scalability needs of Utility Clouds characterized by exa-scale and dynamism, providing an online lightweight technique based on entropy analysis that can operate in a black-box manner across multiple horizontal and vertical metrics. Instead of aggregating multiple metrics into a random variable and tracking their distribution pattern at runtime, our approach establishes response time VS throughput model and continuously update it through online learning. The approach is as lightweight, adaptive and efficient as EbAT for large scale, highly dynamic cloud platforms and services.

New APM product or service vendors such as New Relic [28] and AppDynamics[29] deliver capabilities beyond the traditional fixed threshold based monitoring such as Apdex and dynamic baseline for seasonal workload. The capability offered by our approach can be integrated into these products for improved detection accuracy and longer lead time.

VII. Conclusion and Future work

This paper presents a threshold-less approach for API performance monitoring, which recognizes performance problems by response time deviation from baseline response time - throughput models that are created and continuously updated through online learning. It is designed to be threshold-less, adaptive, lightweight, and independent in order to cate for the requirements of a large-scale, highly dynamic cloud environment. The experiment results preliminarily validate the efficiency and effectiveness of the approach with significantly lowered false positive and false negative. Moreover, with the effective alert reduction algorithm, we can reduce the number of raw alerts significantly, which reduces the SRE's time to resolve problems. Although the results also show longer lead time, it still needs to be compared with prediction model based approaches and the rationale behind also needs to be elaborated. In addition, how to provide more accurate information in anomaly alerts for facilitating further root cause investigation is also an issue to address in the future.

VIII. Acknowledgment

The authors would like to thank all the anonymous reviewers and the people who have given comments on this paper. Ping Wang is the corresponding author.

References

[1] Amazon Web Service, *https://aws.amazon.com/* [Accessed on Jan 12, 2017].

[2] IBM Bluemix Platform, *https://www.ibm.com/cloud-computing/bluemix/* [Accessed on Jan 12, 2017].

[3] Salesforce, *https://www.salesforce.com/* [Accessed on Jan 12, 2017].

[4] C. Wang, S. P. Kavulya, J. Tan, L. Hu and et al, "Performance Troubleshooting in Data Centers: An Annotated Bibliography", *ACM SIGOPS Operating Systems Review*, vol. 47, no. 3, 2013, pp. 50-62.

[5] J. P. Buzen and A. W. Shum, "MASF?Multivariate Adaptive Statistical Filtering", *Computer Measurement Group (CMG)*, 1995, pp. 1-10.

[6] C. Wang, K. Viswanathan, L. Choudur, V. Tal-war and et al, "Statistical Techniques for Online Anomaly Detection in Data Centers", *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 385-392, 2011.

[7] Apdex, http://www.apdex.org/ [Accessed on Jan 12, 2017].

[8] Jim Hirschauer, Apdex is Fatally Flawed, *https://blog.appdynamics.com/product/apdex-is-fatally-flawed/* [Accessed on Jan 12, 2017].

[9] E. Kiciman and A. Fox, "Detecting Application-Level Failures in Component-based Internet Services", *IEEE Trans. on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, 16(5):1027-41, 2005.

[10] W. Xu, L. Huang, A. Fox, D. Patterson and M. I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs" *Proceedings of International Conference on Machine Learning*, 2010, pp. 37-46.

[11] P. Hoogenboom and J. Lepreau, "Computer System Performance Problem Detection Using Time Series Models", *Proceedings of USENIX Annual Technical Conference (ATC)*,1993, pp. 15-32.

[12] C. Stewart and K. Shen, "Performance Modeling and System Management for Multi-component Online Services", *Proceedings of USENIX Symposium on Networked Systems Design and Implementation(NSDI)*, 2005, pp.71-84.

[13] G. Jiang, H. Chen and K. Yoshihira, "Discovering Likely Invariants of Distributed Transaction Systems for Autonomic System Management", *Cluster Computing (Springer)*, 9(4):385-399, 2006.

[14] H. Kang, H. Chen and G. Jiang, "Peer-Watch: a Fault Detection and Diagnosis Tool for Virtualized Consolidation Systems", *Proceedings of ACM International Conference on Automatic Computing (ICAC)*, 2010, pp. 119-128.

[15] K. Shen, C. Stewart, C. Li and X. Li, "Reference-Driven Performance Anomaly Identification", *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009, pp. 85-96.

[16] M. Jiang, M A. Munawar, T. Reidemeister and P. A. Ward, "System Monitoring with Metric-Correlation Models: Problems and Solutions", *Proceedings of ACM International Conference on Automatic Computing (ICAC)*, 2009, pp. 13-22.

[17] J. Lou, Q. Fu, S. Yang, Y. Xu and J. Li, "Mining Invariants from Console Logs for System Problem Detection", *Proceedings of USENIX Annual Technical Conference (ATC)*, 2010, pp. 231-244.

[18] C. Wang, V. Talwar, K. Schwan and P. Ranganathan, "Online Detection of Utility Cloud Anomalies Using Metric Distributions", *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2010, pp. 96-103.

[19] C. Wang, K. Viswanathan, L. Choudur, V. Tal-war and et al, "Statistical Techniques for Online Anomaly Detection in Data Centers", *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2011, pp. 385-392.

[20] C. Wang, "EbAT: online methods for detecting utility cloud anomalies", *Proceedings of the 6th Middleware Doctoral Symposium (MDS 2009) in conjunction with Middleware*, vol.4., 2009.

[21] M. Jiang, M. A. Munawar, T. Reidemeis-ter and P. A. Ward, "Automatic Fault Detection and Diagnosis in Complex Software Systems by Information-Theoretic Monitoring", *Proceedings of IEEE Conference on Dependable Systems and Networks(DSN)*, 2009, pp. 285-294.

[22] K. Ozonat, "An Information-Theoretic Approach to Detecting Performance Anomalies and Changes for Large-Scale Distributed Web Services", *Proceedings of IEEE Conference on Dependable Systems and Networks (DSN)*, 2008, pp. 522-531.

[23] Y. Tan and X. Gu, "On Predictability of System Anomalies in Real World", *Proceedings of IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010, pp. 133-140.

[24] X. Gu and H. Wang, "Online Anomaly Prediction for Robust Cluster Systems", *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2009, pp. 1000-1011.

[25] Y. Tan, X. Gu and H. Wang, "Adaptive System Anomaly Prediction for Large-Scale Hosting Infrastructures", *Proceedings of ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing(PODC)*, 2010, pp. 173-182.

[26] 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", *Proceedings of IEEE Conference on Dependable Systems and Networks (DSN)*, 2008, pp. 452-461.

[27] H. Wu, A. N. Tantawi and T. Yu, "A Self-Optimizing Workload Management Solution for Cloud Applications", *Proceedings of IEEE 20th International Conference on Web Services*, 2013, pp. 483-490.

[28] New Relic, https://newrelic.com/ [Accessed on Jan 12, 2017].

[29] AppDynamics, https://www.appdynamics.com/ [Accessed on Jan 12, 2017].

[30] A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, et al, "Towards automated performance diagnosis in a large IPTV network", *ACM SIGCOMM 2009 Conference on Data Communication*, Vol.39, pp.231-242, 2009.

[31] L. Sommerlade, M. Thiel, B. Platt, et al, "Inference of Granger causal time-dependent influences in noisy multivariate time series", *Journal of Neuroscience Methods*, 203(1):173-185, 2012.

[32] Y. Yu, D. Yao, "Causal Inference Based on the Analysis of Events of Relations for Non-stationary Variables", *Scientific Reports*, 6:29192, 2016.

[33] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, et al , "Detecting novel associations in large data sets", *Science*, vol. 334, no. 6062, pp. 1518 -1524, 2011.