Application-Aware Latency Monitoring for Cloud Tenants via CloudWatch+

Dapeng Liu, Dan Pei [*], Youjian Zhao
Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
Email:{liudp10@mails., peidan@, zhaoyoujian@}tsinghua.edu.cn

Abstract—**A tenant hosted by a cloud platform typically runs a lot of applications, each of which not only has its own service capacity but also differs in revenue or business importance. Thus we argue that it is crucial for a cloud platform to provide fine-grained and application-aware performance monitoring for each tenant, rather than current monitors that can only handle overall metrics. In this paper, we propose CloudWatch+, a tool that focuses on detecting the latency of web-based applications. A cloud platform equipping with CloudWatch+ can automatically learn and distinguish web-based applications it is hosting, and detect latency anomalies for each application based on its own status. Our evaluation using the real data from a cloud platform with over 200 tenants demonstrates that the detection results of CloudWatch+ are more detailed than those of Amazon CloudWatch, which misses most alarms while some tenants' specific applications experience bad performance. Meanwhile, CloudWatch+ is also realtime and light-weight.**

I. Introduction

Cloud computing platforms (e.g., Amazon AWS and Windows Azure) are critical components of the current Internet. These platforms offer attractive properties, such as elastic scaling, integrated management and global accessibility. As such, more and more Internet services are deployed on cloud platforms. For example, Netflix builds its entire online video streaming service on top of Amazon AWS.

As with any Internet applications, web applications are sensitive to various performance metrics, one of which is response latency. For example, 1 second increasing of latency can lead to 11% page views dropping and 7% customer conversations lost according to [1]. Therefore, latency monitors are considered to be indispensable components of the cloud platforms hosting web applications. There have been a suite of tools provided within a cloud, such as Amazon CloudWatch [2] and Windows Azure Metrics [3]. Besides, many solutions from SaaS-based third parties are available as well, such as RightScale [4] and New Relic [5]. Beyond monitoring, Amazon CloudWatch is able to automatically trigger auto-scaling (provisioning more cloud instances) when performance degradation has been found.

Despite the efforts of these commercial monitors, the metrics they gathered are all aggregated at either an instance level or a tenant level (e.g., Amazon CloudWatch), and measured with minimum, average, and maximum [6], [7], [8], [9]. . This fundamentally limits the effectiveness of the web latency monitoring for tenants, because (1) the applications of a tenant could differ in their importance, e.g., VIP users' applications and revenue related applications are more cared by tenants; (2) a single application might contribute to only a small part of the total requests [10], thus their latency anomalies (if any) cannot result in an obvious symptom of the overall latency (both average and 90^{th} percentile). Although the maximum can reveal any latency anomaly, it is commonly believed to be too noisy and triggers a large number of false positives, thus is rarely used in practice. As a result, tenants cannot be informed for the performance degradation of one or more specific important applications.

While overall performance metrics is coarse-grained, monitoring and alarming on each individual URL (raw data in typical web access logs) is also ineffective. This is because an application may produce a huge number of unique URLs, and a failure happens on an application, the alarms on individual URLs would be overwhelming. For example, in one-day data set (after sampling by $1/60$), there are still more than 70,000 suspicious records with different URLs when the latency threshold is set as 1.5 seconds. It's an intractable task for tenants to manually figure out which applications are suffering serious performance problems from these URLs.

To address the above dilemma, we propose an application-aware web latency monitor for cloud tenants. Its granularity is fine enough for tenants to identify the poor applications. We name our tool as **CloudWatch+**, and it can be considered as an application-aware version of Amazon CloudWatch.

The architecture of CloudWatch+ is shown in Fig. 1. The input data is web access logs, recording the timestamp, URL, response latency, etc., for every visit. The data is then handled by two main components of CloudWatch+ in sequence: (1) *uCluster* is a module that automatically classify a large number of web access records into different applications based on URL clustering. Its high level idea is exploiting the URL pattern for each application, e.g., common fields of URLs. (2) *Anomaly detector* calculate the latency and workload of every time slot for each application. In this paper The latency is measured with the 90^{th} percentile, which is a common measurement of performance; the workload is measured with query per seconds (QPS). The anomaly detector determines anomalies by comparing the latency of each application to a threshold. Beyond that, it can also estimates the capacity of applications. Based on this information, it can infer whether a latency anomaly is caused by overload or not, a rather important suggestion for tenants to make accurate decisions of whether to scale up (e.g., purchasing more instances from the cloud).

CloudWatch+ has the following advantages:

- *URL based web application identification and no prior knowledge.* CloudWatch+ exploits URLs, common identities of web access, to distinguish applications.

[*] Dan Pei is the corresponding author.

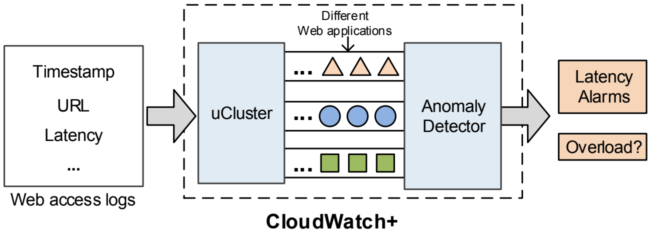


Fig. 1: Architecture of CloudWatch+.

Moreover, it does not know in advance what applications are run by tenants, nor does it know the URLs pattern (i.e., which fields of a URL represent applications and which ones represent parameters). CloudWatch+ simply monitors web access logs and automatically learns the information.

- *Online and Realtime*. Since the anomalies of application latency can severely cause damages to users' experience, CloudWatch+ is designed to be online and realtime. Thus, tenants can be warned as soon as possible, and then take actions to stop further loss.
- *Light-weight*. CloudWatch+ is also light-weight in space and CPU complexity so that it can be easily deployed as an online tool in the cloud, that hosts a potentially large number of tenants and applications.

We evaluate CloudWatch+ using the real data from a cloud platform hosting over 200 tenants. The results show that, CloudWatch+ is able to identify the anomalies of each application separately, while almost all these anomalies are missed by Amazon CloudWatch. In addition, the storage and CPU overhead are very low even when monitoring and detecting a large amount of access records.

The remainder of the paper is organized as follows. We present the observations and intuitions from the analysis of real data in Section II. The design of the online URL clustering algorithm and the corresponding results are described in Section III and Section IV respectively. We introduce our anomaly detection method based on uCluster in Section V, and evaluate CloudWatch+ in Section VI. We discuss related work in Section VII before concluding this paper in Section VIII.

II. Data Set and Offline Analysis

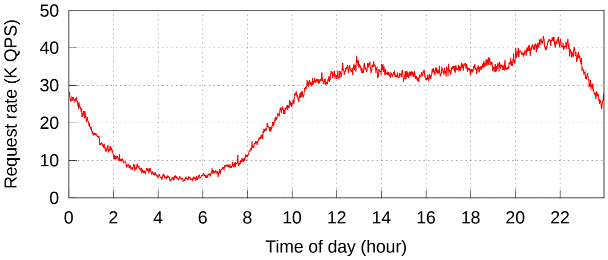


Fig. 2: Overall request rate at the cloud data center.

To gain more insights into the web access in a real cloud, before developing our methods, we first introduce the one day worth of web access log obtained from one cloud provider. It serves more than 200 tenants, and the overall workload reaches 42K QPS during peak-hour around 22:00 (as shown in Fig. 2). As some tenants only receive a small amount of workloads, we focus on the top 64 most visited tenants in the rest of the paper. These tenants provide services such as online social network, news portals, picture sharing, video sharing, music streaming, document sharing.

Our data set is collected at the load balancers of the cloud, with a sampling rate of $1/60$[1]. In total, we obtained 12.5G worth of data with 33 million records. In each record, there are multiple fields, three of which are used by CloudWatch+, i.e., $\{$*Request-Timestamp, URL, Latency*$\}$. The latency refers to the time between when a request is forward to a server by a load balancer and its response returns back. This measurement is adopted similarly by commercial Cloud performance monitoring tools, such as Amazon CloudWatch [2], but they aggregate the latency at an instance level or a tenant level. Our goal in this section is to investigate whether and how we can monitor the latency at a fine granularity.

A. A Motivating Example

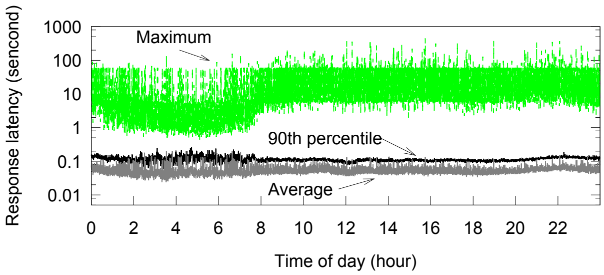


Fig. 3: The maximum, 90^{th} percentile, and average of latency during a day.

Within the data set, a particular tenant (an online social network service) received a lot of user complaints about the long latency for several applications from 23:30 to 00:30. We measure the overall metrics of latency for this tenant, like existing monitors. Fig. 3 plots the three metrics derived from the data set. We can see that the average and 90^th percentile of latency are always under 0.25s, representing a good performance. On the other hand, the maximum fluctuates significantly all the time, thus is seldom used for raising performance alarms. The result shows that these overall metrics are all infeasible to capture the real user perceived problems, and proves that the coarse-grained monitoring is ineffective.

Motivating by the above example, we intend to design an application-aware latency monitoring method for web applications in this paper. The high level idea is intuitive, that is distinguishing web applications, then measuring and detecting their latency separately. In order to identifying web applications generally, we exploit a very common field in web access logs, i.e., URL.

B. Observations from the data set

We now present some observations from offline analysis of the data set. These observations both highlight the challenges and offer the intuitions of designing such a method.

Observation 1: *Most tenants have a large number of unique URLs.* Fig. 4 shows the number of unique URLs

[1] Unless otherwise noted, the workload in this paper refers to the one in the sampled data set.

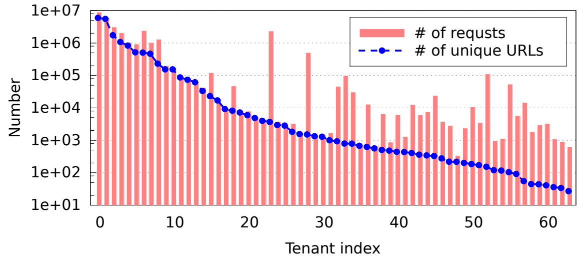


Fig. 4: Then number of requests and unique URLs per tenant.

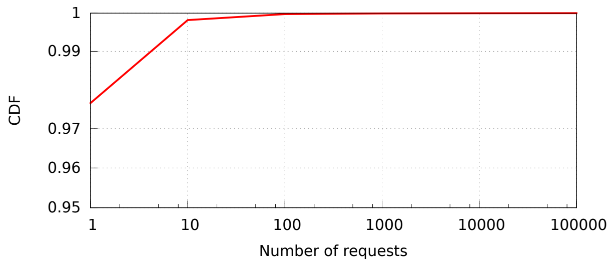


Fig. 5: The CDF of the request number of each URL.

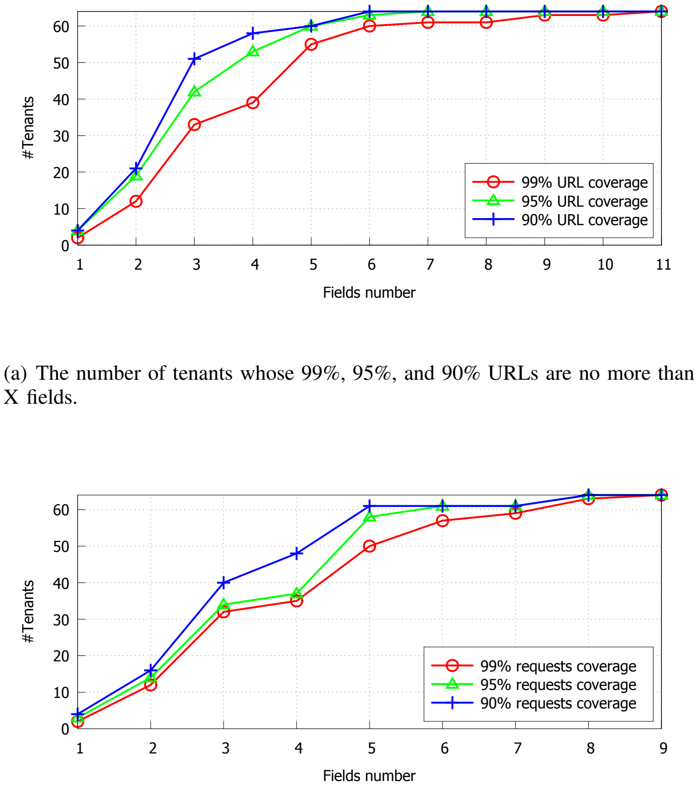
generated by each tenant, as well as the number of requests received by each tenant. The tenants are ordered along X axis by the number of unique URLs in descending order. As shown in Fig. 4, the number of unique URLs owned by each tenant ranges from tens to millions, and the number of requests per tenant ranges from hundreds to millions (after sampling). We also notice that for many tenants (e.g. the first tenant), the numbers of its unique URLs and requests are both huge. This is because the URLs contain some conversation-specific parameters such as date time and transaction ID, thus every request will generate a different URL even if they visit the same application.

Observation 2: *Most URLs are visited only once during a day.* Fig. 5 shows the CDF of the request number of each URL. It shows that 97.7% of URLs are visited only once during that day. Only 0.007% URLs received requests more than 1000, such as the home pages or static images.

Observation 3: *Most URLs have only a few fields.* As illustrated in Fig. 6(a) , for all the 64 tenants, 99% of their URLs are up to 11 fields, and for 61 of them, 99% of their URLs are at most 7 fields. By further investigating, we find that the tenants with URLs of a few fields mostly provide simple index pages (e.g., /news/20130102.html) or static content caching (e.g., /banner/top.gif). Similarly, requests are more likely to hit the URLs with less fields. As shown in Fig. 6(b), for all the 64 tenants, 99% of their requests hit the URLs with no more than 9 fields, and for 57 of them, URLs with less than 7 fields absorb 99% requests. This observation also suggests that the number of common fields shared by different URLs is even less.

C. Challenges and Intuitions

To detect the latency anomaly at web application granularity, we first identify applications via URLs. However, this is challenging due to the following two aspects. First, the URLs of the same application always carry *parameters*



(a) The number of tenants whose 99%, 95%, and 90% URLs are no more than X fields.

(b) The number of tenants whose 99%, 95%, and 90% requests hits the URLs that are no more than X fields.

Fig. 6: The distribution of the number of URL fields.

with it, yet eliminating them is intractable due to the URL rewriting technology[11]. Specifically, the parameters can exist not only in the last field after "?" as standard URLs appear, but also anywhere of URLs. For example, "/news/0001" is rewritten from "/news?id=0001". Second, as Observation 1 shows, a large number of user requests visit unique URLs (e.g., same application, but with different parameters). This renders manually analyzing and identifying applications impractical and raises a requirement of an automatical URL clustering method for application identification.

Based on the data analysis, we leverage the following intuitions to track the above challenges and design CloudWatch+.

Intuition 1: *A URL pattern with a parameter field can generate more different URLs.* Each URL pattern can generate a different number of specific URLs sharing several common fields. Particularly, the URL pattern with a parameter field can result in much more unique URLs than the one without parameter field. For example, "/news/*parameters*" pattern can have many specific and unique URLs such as "/news/0001", "/news/0002",..., "/news/9999". On the other hand, another URL pattern "/user/*application*" can only have a handful of unique URLs such as "/user/blog" and "/user/info". Furthermore, the number of requests hitting URLs with parameters are much less because the requests are dispersed by different parameter values.

Intuition 2: *URL pattern of the same application is hierarchical by nature.* Although URLs can be encoded very flexibly in reality, the practical URL patterns are hierarchical. This is because web servers typically organize files (for static content) and codes (for dynamic scripts) in a hierarchical folder structure. This implies that the clusters of URLs are hierarchical and a cluster can be represented by the common fields of the URLs belong to it.

III. Online URL Clustering Algorithm

In this section, we first introduce the limitations of some existing clustering methods. Afterwards we formalize our problem and describe our clustering algorithm.

A. Limitations of Existing Clustering Methods

There are many clustering technologies used for partitioning a data set, such as classical k-means and hierarchical clustering. However, it is still faced with many challenges to apply them in online monitoring for URL clustering: (1) Their high complexity is unacceptable for online monitoring a huge quantities of data, e.g., k-means is NP-hard [12] and hierarchical clustering is $O(n^3)$ and $O(n^2)$ for special cases [13]. (2) The clustering terminal conditions have significant impacts on the clustering results, yet it is difficult to manually determine those values intuitively (e.g., k value in k-means). (3) Assumption of a complete data is inflexible when the handling incremental data, such as dealing with the change of URL patterns caused by the service updates.

B. Problem Formalization

For convenience of later discussions, we first review some basic concepts used in clustering.

- x, an element of the data set, and is denoted by a vector: $x = \{v_1, v_2 ... v_n\}$.
- $\mathbf{D}(x, y)$, the distance function between two elements x and y (e.g., Euclidean distance and Hamming distance).
- $\mathbf{LC}(\mathcal{X}, \mathcal{Y})$, the linkage criteria specifying the distance between clusters $\mathcal{X}$ and $\mathcal{Y}$ (e.g., single-linkage $\min\{\mathbf{D}(x, y)|x \in \mathcal{X}, y \in \mathcal{Y}\}$ and complete-linkage $\max\{\mathbf{D}(x, y)|x \in \mathcal{X}, y \in \mathcal{Y}\}$).

Hence, all the elements are sequentially merged into clusters according to the distance functions and strategies used in different methods.

Now we formalize the URL clustering problem. Each URL u has a set of fields split by a delimiter ("/" most frequently used). Particularly, if there is a "?" appearing in the last field, we remove the part after the "?", aiming at preliminarily filtering out the conspicuous parameters of standard URLs. Then the fields of u is denoted by $F(u) = \{f_1, f_2, ..., f_i\}$; each field f is indexed based on its position in u. For example: $F(\text{forum/userInfo?12416}) = \{\text{forum,userInfo?}\}$.

The distance between URLs can be measured with the number of common fields. Therefore, the distance function $\mathbf{D}$ is defined as follows:

$$\mathbf{D}(u_1, u_2) = \begin{cases} \infty & \text{, if } F(u_1) \bigcap F(u_2) = \varnothing \\ \frac{1}{|F(u_1) \bigcap F(u_2)|}, & \text{otherwise} \end{cases} \tag{1}$$

This indicates that the more common fields a pair of URLs have, the closer they are to each other. Note that whether the common fields are consecutive is not critical in the definition. This yields the ability to handle the parameters in the middle part of a URL.

C. uCluster Algorithm

To overcome the aforementioned challenges, we design *uCluster*, a URL clustering algorithm. By leveraging our observations and intuitions, uCluster differs URL clustering problem from general cases. It aims at clustering URLs based on their structural similarity and eliminating parameters of applications. Our design of uCluster possesses the following advantages: (1) low complexity $O(n)$; (2) intuitive arguments of clustering, which can be derived from analyzing data itself; (3) the online manner without assuming complete data.

Firstly, We introduce two concepts, *virtual clusters* and *final clusters*, to aid in eliminating parameters in URLs and achieving the ability of online process. Virtual clusters are the intermediate clusters generated according to the distance between URLs, and they are the candidates into which a URL would be finally grouped. Final clusters are the resultant clusters, and also represent the applications we identified. They are converted from virtual clusters based on the number of subclusters (as we describe later).

A virtual cluster is denoted by $\mathcal{V}$, and the set of common fields shared by the URLs in $\mathcal{V}$ is denoted as $F(\mathcal{V})$, where $F(\mathcal{V}) = \{\sum \cap F(u)|u \in \mathcal{V}\}$. All the virtual clusters are hierarchically structured as a tree, and the level (or depth) of $\mathcal{V}$ is denoted as $l(\mathcal{V})$. The nature of each $\mathcal{V}$ is $|F(\mathcal{V})| \geqslant l(\mathcal{V})$, which means that the number of the common fields shared in $\mathcal{V}$ must be no less than the level of $\mathcal{V}$. This nature can also be interpreted as $\forall u_1, u_2 \in \mathcal{V} : \mathbf{D}(u_1, u_2) \leqslant 1/l(\mathcal{V})$, that is as $\mathcal{V}$ goes deeper in the tree, the distance between the URLs in $\mathcal{V}$ should be shorter. As Observation 3 shows, the fields number of most URLs are always only a few, indicating the maximum level of the tree, denoted as L, is small.

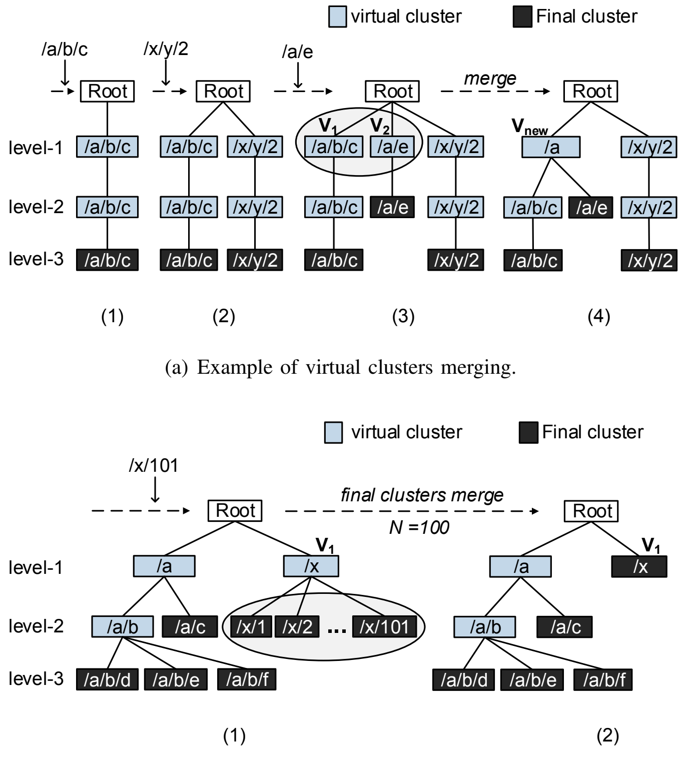
The linkage criteria between $\mathcal{V}_1$ and $\mathcal{V}_2$ is defined as:

$$\mathbf{LC}(\mathcal{V}_1, \mathcal{V}_2) = \frac{1}{|F(\mathcal{V}_1) \bigcap F(\mathcal{V}_2)|} \tag{2}$$

And the conditions when $\mathcal{V}_1$ and $\mathcal{V}_2$ can be merged are that (1) $\mathcal{V}_1$ and $\mathcal{V}_2$ have the same direct ancestor and they are at the same level, $l(\mathcal{V}_1) = l(\mathcal{V}_2) = l$ (to guarantee the hierarchical structure); (2) the new merged cluster should also satisfy the nature of virtual cluster, that is $\mathbf{LC}(\mathcal{V}_1, \mathcal{V}_2) \leqslant \frac{1}{l}$.

When a pair of virtual clusters, $\mathcal{V}_1$ and $\mathcal{V}_2$, have been merged into a new virtual cluster $\mathcal{V}_{new}$, we have $F(\mathcal{V}_{new}) = F(\mathcal{V}_1) \bigcap F(\mathcal{V}_2)$. $\mathcal{V}_{new}$ is still at the same level as $\mathcal{V}_1$ and $\mathcal{V}_2$.

Fig. 7(a) gives an example of how virtual clusters are constructed and merged (final clusters in Fig. 7(a) will be introduced later): (1) When a URL "/a/b/c" arrives, it makes three virtual clusters on each level, as it has three fields. These virtual clusters are all initially represented by F(/a/b/c); (2) Similarly, a URL "/x/y/e" constructs three virtual clusters in the same way, but merging conditions are not hold here and no clusters are further merged; (3) After "/a/e" arrives and construct two virtual clusters, $\mathcal{V}_1$ and $\mathcal{V}_2$ (in the circle) satisfy merging conditions, where $F(\mathcal{V}_1) = F$(/a/b/c) and $F(\mathcal{V}_2) = F$(/a/e). Specifically, $\mathcal{V}_1$ and $\mathcal{V}_2$ belong to the same direct ancestor (root), $l(\mathcal{V}_1) = l(\mathcal{V}_2) = 1$, and $\mathbf{LC}(\mathcal{V}_1, \mathcal{V}_2) = \frac{1}{|F(\mathcal{V}_1) \bigcap F(\mathcal{V}_2)|} \leqslant 1$. (4) So that $\mathcal{V}_1$ and $\mathcal{V}_2$ are merged into $\mathcal{V}_{new}$, and $F(\mathcal{V}_{new}) = F$(/a). Corresponding children clusters change their hierarchical relations from original ones to the new cluster.



(a) Example of virtual clusters merging.

(b) Example of final clusters merging.

Fig. 7: The work flows of virtual clusters and final clusters.

Through this merging strategy, we can obtain all the virtual clusters, and now we need to determine which of them are the final clusters, i.e., applications we identified (We will use cluster and application interchangeably). Note that the final clusters are generated in parallel with virtual cluster merging. The conditions for final clusters are two aspects. First, all the leaf nodes are labeled as final clusters, as shown in Fig. 7(a) and Fig. 7(b). Second, according to the intuition 1, the numbers of unique URLs produced by the URL patterns with and without parameters are different. Therefore, we set a boundary threshold, denoted as N, to determine the fields of parameters. In particular, if a virtual cluster $\mathcal{V}$ has more than N sub-clusters (i.e., children clusters), including both virtual and final clusters, we deem that these sub-clusters are generated by URL parameters, and should be eliminated. Thus $\mathcal{V}$ itself turns into a new final cluster. An example of final clusters merging is shown in Fig. 7(b). After the arrival of "/x/101", the sub-clusters number of $\mathcal{V}_1$ increases to 101, and exceeds the given threshold $N = 100$. As a result, the sub-clusters of $\mathcal{V}_1$ are removed, and $\mathcal{V}_1$ becomes a new final cluster. We can see that this heuristic merging strategy can find out the fields derived from URL parameters without assumptions of their positions.

In general, clustering methods always face multiple clusters that have the same optimal distance, leading to different clustering results, uCluster is no exception. Worse still , this can generate false clusters in the scenario of online URL clustering, as the results depend on the arrival sequence of URLs. For example, if the first two emerging URLs are "/news/100" and "/image/100", they will be merged and generate a false virtual cluster, i.e., "//100", which turns out to be a parameter field, instead of an application field. We solve this problems based on the observation that the number of unique URLs matching a real application cluster (e.g., "/news/*") is much larger than those match a related false cluster (e.g., "//100"). Therefore, when choosing clusters for merging, we do it in descending order of the number of the unique URLs contained by each virtual cluster. This trick would distribute most URLs to the real application clusters. As a result, wrong clusters are always ignored when a multiple clusters can be merged. Only a few incorrect merging in the initial phase have trivial influence on the accuracy when dealing with a huge volume of data.

Finally, we propose Algorithm 1 based on uCluster. It's easy to see that its complexity is $O(N\textbf{.}L\textbf{.}n)$, where L and N are both small constants, so that the complexity is just $O(n)$.

Algorithm 1 uCluster (URL u, L, N)

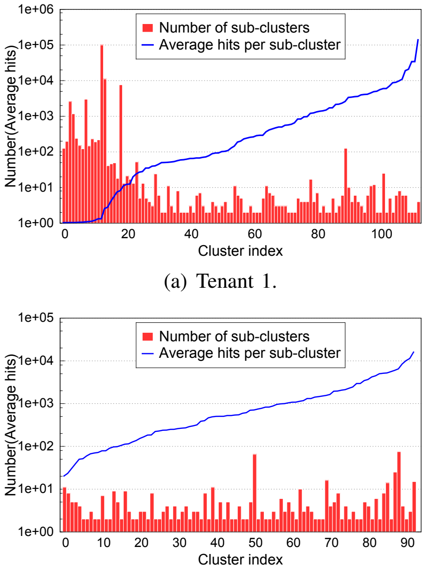
```
$current = root$
for $d = 1$ to $min(L, |F(u)|)$ do
  $merge\_flag == false$
  if $current$ is not $FINAL\_CLUSTER$ then
    for each $CLUSTER$ $c$ in $current$ DESC do
      if $|F(c) \bigcap F(u)| \geq d$ then
        $current = c$
        $merge\_flag = true$
        $F(c) = F(c) \bigcap F(u)$, break
      end if
    end for
    if $merge\_flag == FALSE$ then
      if $|F(current)| == N$ then
        set $current$ $FINAL\_CLUSTER$, break
      else
        $VIRTUAL\_CLUSTER$ $v = F(u)$
        $current.add\_virtual\_cluster(v)$
        $current = v$
        if $d == min(L, |F(u)|)$ then
          set $current$ $FINAL\_CLUSTER$
        end if
      end if
    end if
  end if
end for
```

IV. Arguments Selection and Clustering Results

In this section, we show that the two arguments of the uCluster algorithm, N and L, are quite convenient to select based on the characteristics of data. Our arguments selection methodology uses, but is not restricted to, the data set we collected.

First of all, the argument N has a simple and practical implication, that is the maximum number of the applications contained in each application directory. Therefore, it can be easily estimated by tenants. One alternative way is counting the the maximum number of the accessible files included by each application directory. This differs with those unintuitive arguments used by other clustering methods, such as the k value in k-means algorithms.

The administrator from one of the tenants suggest that $N = 100$ is large enough , and our evaluation proves it works well in our data set. Our evaluation methodology is based on the observation 1, i.e., the number of clusters generated by parameter pattern is large, in the meantime, the average hits number of them is small since the requests are dispersed by the parameters of a wide value range. We run uCluster on 64 tenants without the constraint of N to get all the number of sub-clusters, as well as their average hits. Due to the page limitation, we only show two typical tenants here. As shown in Fig. 8, X axis is the cluster index and Y axis represents both the number of sub-clusters and average hits in log scale. Fig. 8(a) (tenant 1) indicates that, for the clusters with a huge number of sub-clusters, the average of hits on these sub-clusters falls below 10. Therefore, they are more likely to be parameter patterns, as implied by intuition 1. We can see that if we draw a horizontal line at Y equals 100 (i.e., $N = 100$), most of the parameter patterns would be merged, and other application pattern would remain. In addition, Fig. 8(b) illustrates that tenant 2 does not have any obvious parameter pattern and $N = 100$ also works well with this situation, as it treat all these sub-clusters as applications.



(a) Tenant 1.

(b) Tenant 2.

Fig. 8: The relationship between sub-clusters number and average hits per sub-cluster without constraint of N (L=5).

As for the argument L, it can be derived from data analysis. We run uCluster with $N = 100$ and list the results of top 10 most visited tenants. As Fig. 9 shows, uCluster groups massive unique URLs (from 1.6×10^5 to 6×10^6) of the 10 tenants into tens to hundreds of clusters. The number of clusters increases as L first, and then remains stable after L exceeds 5. This also proves the observation 3, i.e., most URLs have only a few fields. In fact, the number of clusters also represents clustering granularity. So $L = 5$ is proper for our data set, as a larger L would not improve granularity further.

An automatic program based on the above selection methodology is relatively easy to implement. In the interest of space, we leave the detailed discussion on these topics as our future work.

V. Latency Anomaly Detection

We first introduce the limitations of detection based on overall metrics of latency. Then we describe our detection methods.

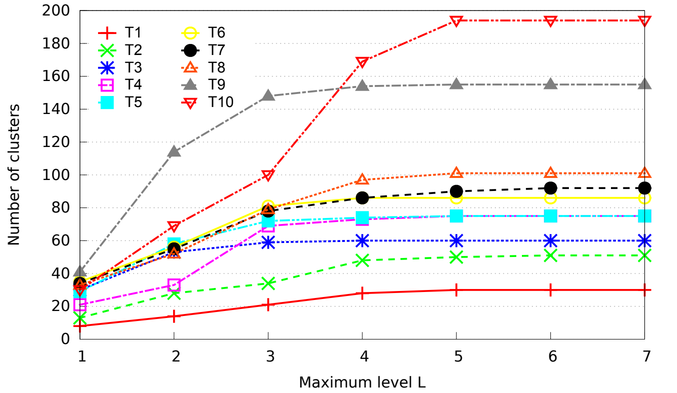


Fig. 9: The clustering results of top 10 most visited tenants. L values from 1 to 7 under the condition of $N = 100$.

A. Limitations of Monitoring Overall Metrics

In this paper, we focus on the 90^{th} percentile of latency (latency^{90} for short), a common measurement for performance. When monitoring overall latency^{90} for a tenant, alarms are triggered if more than 10% of the tenant's volume are served in more than a threshold of time. Under this situation, if an application tries to raise alarms in the face of overall latency^{90}, while other applications remain normal[2], the anomalous volume of the application needs to exceed 10% of the tenant's volume, which is always larger than its own 10% volume. This actually degrades the threshold of 90^{th} percentile for the application, and even makes it impossible for some applications to be detected by itself as their volume accounts for less than 10% of total volume. The degraded percentile threshold of application α is denoted as P_α, and suppose the volumes of α and the tenant hosting it are V_α and V_t respectively, then P_α can be calculated as:

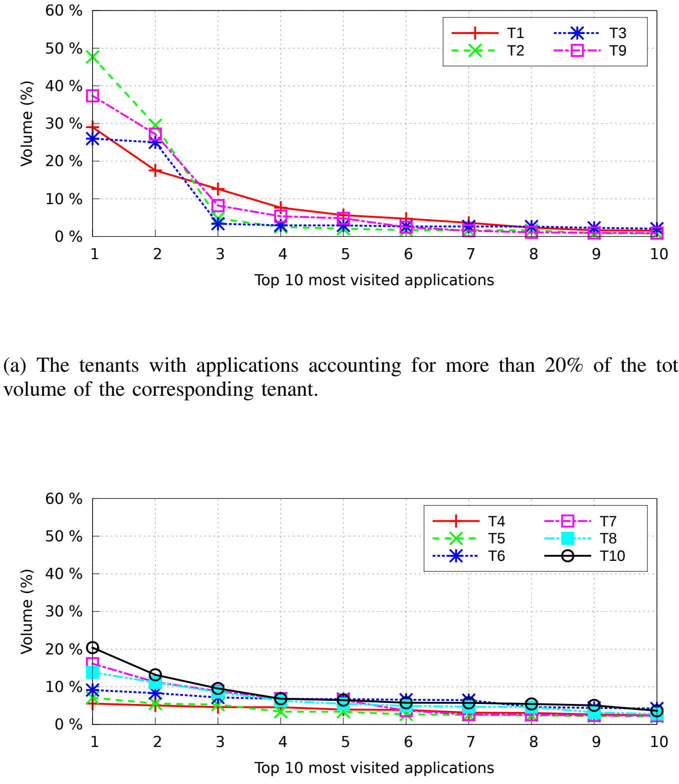
$$P_\alpha = \begin{cases} -\infty & \text{, if } V_\alpha/V_t < 10\% \\ 100\% - \frac{10\%}{V_\alpha/V_t} & \text{, otherwise} \end{cases} \tag{3}$$

V_α/V_t is the volume proportion of α in its tenant. Fig. 10 shows the volume proportion of the top 10 most visited applications of the top 10 tenants. Fig. 10(a) shows that there are 4 tenants with the applications receiving more than 20% (up to 47%) of their total requests, and Fig. 10(b) shows another 6 tenants whose applications all account for less than 20% of their volume. As a result, even for application 1 of tenant 2 in Fig. 10(a), that contributes to the most proportion 47%), its degraded percentile equals to 78%, implying the application needs 22% anomalous volume to catch the attention of detector on overall latency^{90}. Worse still, for most applications of these 10 tenants, their proportions are all less than 10%, so than they cannot be detected in the face of overall latency^{90}.

B. Latency Detection on Applications

Benefit from uCluster, web applications are separated and the latency^{90} can be measured for each application. This

[2] For simplicity, we only discuss the situation when only a single application is anomalous. This provides the worst case for monitoring on overall latency.



(a) The tenants with applications accounting for more than 20% of the total volume of the corresponding tenant.

(b) The tenants without applications accounting for more than 20% of the total volume of the corresponding tenant.

Fig. 10: The volume distribution of the top 10 most visited applications of the top 10 tenants. The applications are identified by uCluster with $N = 100$ and $L = 5$.

fundamental improvement gains the ability of application-aware latency anomaly detection for cloud tenants. Meanwhile, as each virtual cluster can maintain the metrics aggregated from all its sub-clusters, CloudWatch+ can conduct flexible anomaly detection at different granularity of applications, or even detect the overall metrics (i.e., using the data of root clusters).

Beyond the improvement of granularity, we also want to answer this question: “Is a latency anomaly caused by overload?”. This is an important problem for cloud tenants, as they need to decide wether their capacity is inadequate and more instances are required. However, a latency anomaly might be caused by many factors, such as software bugs and hardware failure. We resolve this problem by keeping a track of the workload that has been well served by each application of tenants, aiming at estimating the capacity of different applications. So that when an anomaly happens to a certain application, we can determine overload by simply checking wether the current workload exceeds the historical capacity.

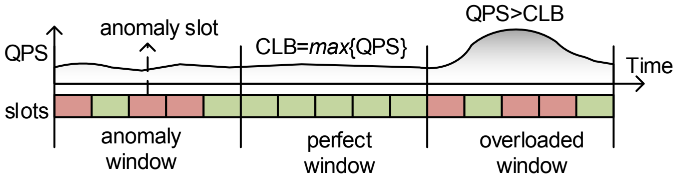


Fig. 11: Detecting latency anomaly and deciding whether overload happens. CLB is short for capacity lower bound.

The details of our anomaly detection are shown in Fig. 11. Firstly, the latency^{90} and workload (measured by query per second, or QPS) of each application are calculated for every slot (10s is used in this paper). By setting a threshold of latency (as Amazon CloudWatch does), a slot is called *anomaly slot* if its latency^{90} exceeds the threshold. To avoid transient anomalies trigger alarms more than frequently, we introduce *window*, formed by W consecutive slots (in Fig. 11, $W = 5$). If there are n anomaly slots in a window, we call the window *anomaly window*, and an alarm is eventually raised. The parameters n and W decide the sensitiveness of detection. Meanwhile, we maintain and update the *capacity lower bound* (CLB) for each application. CLB records the largest average QPS of *perfect window*, that is the window without any anomaly slot. CLB conservatively estimates the capacity of an application so far. An anomaly window is called *overloaded window* if its average QPS exceeds CLB. Then we can calculate the percentage of the overloaded windows within an anomaly period, defined as *overload ratio*. It gives the possibility of that an anomaly period is caused by overload.

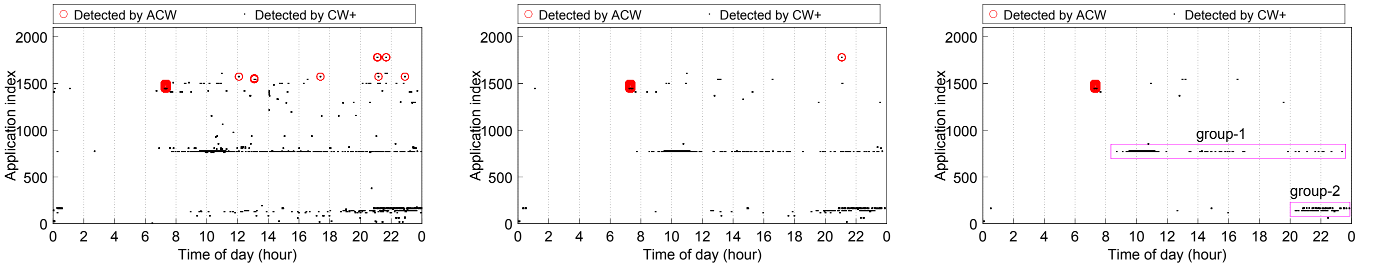
VI. Evaluation

In this section, we evaluate CloudWatch+ using the one-day data set from a real cloud. The data set has been described in Section II. The prototype of CloudWatch+ is implemented with 2700 C++ lines.

We begin by comparing the detection result of CloudWatch+ (CW+) to that of Amazon CloudWatch (ACW), which provides only tenants level monitoring. Both methods use 1.5s as the latency threshold, and are compared under different n/W. In addition, CloudWatch+ is running with $N = 100$ and $L = 7$ according to the Section IV. The results are shown in Fig.12. According to the detection result of ACW, all the 64 tenants seem in good performance except only a few transient anomalies.

After further investigating the anomalous periods identified by ACW (as listed in Table. I), we find out that they belong to four tenants. Moreover, each of these anomalies is caused by only one application. These anomalous applications all account for a considerably large part of their tenants' volume, ranging from 24.2% to 99.7%. This is also the reason why ACW can detect them using overall latency^{90}. Another explanation of that is the degraded percentiles of these applications are close or even equal to the original 90^{th}. The above result proves that monitoring on overall metrics can only detect the anomaly of volume-dominate applications, yet most other applications are invisible for it.

On the contrary, Fig.12 shows that CW+ is able to detect at an application granularity. Specifically, as the sensitiveness threshold n/W increases, less severe anomalies are ignored and the the number of alarms (anomaly periods) are reduced from 472 in Fig.12(a) to 123 in Fig.12(c). Here, n/W can be set by tenants according to their requirements.



(a) $n/W = 3/10$.

(b) $n/W = 4/10$.

(c) $n/W = 5/10$.

Fig. 12: The latency^{90} anomalies of one-day period identified by Amazon CloudWatch (ACW) and CloudWatch+ (CW+) respectively, based on 1.5s as threshold. Y axis represents the index of 2100 applications of top 64 tenants. Each anomaly point indicates an anomaly window of 100s duration. Note that since ACW cannot distinguish applications, i.e., its alarms are at tenant level, the anomaly of ACW will cover all the tenant' applications along Y-axis. (Best viewed in color.)

Tenant ID	# APP	#Anomalous APP	Anomalous APP volume(%)	Degraded percentile
11	57	1	65.8%	85^{th}
15	14	1	24.2%	59^{th}
17	2	1	99.7%	90^{th}
28	3	1	96.6%	90^{th}

TABLE I: The information of 4 anomaly periods detected by ACW with $n/W = 3/10$. APP is short for applications.

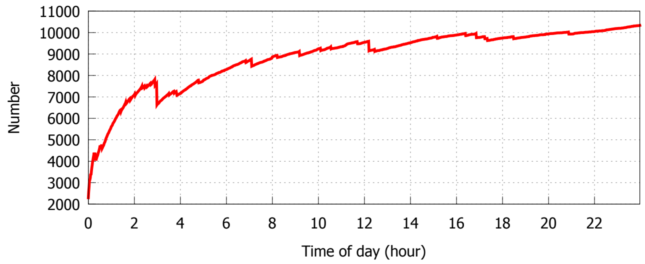


Fig. 13: The number of clusters, including both virtual and final clusters, during the processing of one-day data.

Determining false positive rate is challenging, as investigating all the anomalies would take a huge amount of effort. Instead, we study two notable groups of anomalies detected by CW+, and shows that they are reasonable. As shown in Fig.12(c), applications of group-1 is a upload module of a file sharing service (tenant 1), and the anomaly is caused by the saturated bandwidth during three peak periods of a day; group-2 is a signing up application of an online social network (tenant 2), and a large number of users' visiting causes their database overloaded.

We also list the anomaly periods with overload ratio $\geq 50\%$ for tenant 1 and 2. As shown in Table II, the longest anomaly period comes from tenant 2 and lasts for 6100s, corresponding to group-1 in Fig.12(c). Worse still, the overload ratio of that period is 73.61%, a rather high value that indicates the capacity of tenant 2 is not enough for this application. As for tenant 1, its two applications are overloaded at nigh, especially the overload ratio of "/sign/add?" reaches 100%. The result has been confirmed by the tenant, saying that this application always suffers from flash crowd at midnight, as there are incentives for users who sigh up first of a day.

As for the overhead of CloudWatch+, the storage consumption of the major data structure can be measured by the number of both virtual and final clusters. As shown in Fig. 13, when processing all the over 200 tenants together, the number of clusters increases to around 10,000, which is trivial for commercial servers. In addition, it takes CloudWatch+ merely 875s to process all the one-day data, including both clustering and detecting. This effectiveness can lead to a realtime ability.

VII. Related Work

There are many previous literatures focusing on the performance management of cloud platform. One main thread comes from the solutions of commercial cloud providers , such as AWS CloudWatch [2], Windows Azure Metrics [3], as well as SaaS-based third parties, such as RightScale Monitor [4] and New Relic [5]. They provide monitoring metrics in a wide range, including both physical and service status. However, all of them are at the tenant or instance granularity, thus inefficient to detect the anomalies of a tenant's specific applications as we previously introduced. Besides, some performance monitoring methods are built on traditional platforms. For example, [14] is based on HP Mercury Diagnostics [15], trying to correlate the hardware metrics of a single server (e.g., CPU utilization and network bandwidth) and application performance; [16], [17] are both based on coarse grained and pre-defined applications of web service (e.g., css, js, php), while CloudWatch+ does not assume any prior knowledge about the applications and automatically clusters the URLs into applications.

Besides, several anomaly (or change) detection methods [18], [19], [20], [21] have been proposed and applied in different fields. In this paper, rather designing a complicated detection methods itself, we focus on improving the granularity of detection. Therefore, our work is complementary to these methods, and they can be applied upon CloudWatch+.

VIII. Conclusion

In this paper, we propose CloudWatch+, an application-aware latency monitoring tool for web applications. A cloud platform can use CloudWatch+ to automatically learn web

Tenant ID	Application	Interval	Avg latency^{90}	CLB QPS	Avg QPS	MAX QPS	Overload ratio
1	/sign/add?	00:01:22-00:02:52 (100s)	2.73	2.43	3.50	5.30	100.00%
1	/pad/commit/message/get_all?	21:48:22-21:49:52(100.00s)	1.26	0.71	0.69	1.20	50.00%
2	/i/submit	08:49:42-10:31:12 (6100s)	3.21	0.13	0.21	1.40	73.61%
2	/submit?	09:06:22-09:07:52 (100s)	13.49	0.45	0.41	0.70	50.00%
2	/submit?	09:09:42-09:11:12 (100s)	15.22	0.45	0.57	1.10	70.00%
2	/submit?	09:23:02-09:24:32 (100s)	23.41	0.45	0.56	0.90	90.00%
2	/i/submit	14:11:22-14:14:32 (200s)	4.10	0.13	0.18	0.50	50.00%
2	/i/submit	16:34:42-16:36:12 (100s)	1.96	0.13	0.25	1.10	60.00%

TABLE II: Anomaly periods with overload ratio$\geq 50\%$ for tenant 1 and 2 ($n/W = 5/10$). Each record includes the tenant ID, name of anomalous application, anomaly interval time, the average latency^{90} of the interval, the average and maximum QSP of the interval, as well as CLB and overload ratio.

applications it hosts, and detect latency anomalies at the granularity of application. The benefits of CloudWatch+ to a cloud and its tenants are two-fold. First, the business-critical applications of a tenant, regardless of their volume, can be monitored based on their own metrics (e.g., the 90^{th} percentile of latency). Such service is not currently available in performance monitoring services provided by commercial clouds. Second, tenants can figure out whether the current anomalies are due to overload (as suggested by CloudWatch+). Then they can take right actions to stop further loss, such as purchasing more instance to increase capacity or optimize applications to reduce their consumptions.

Our evaluation using real data from a cloud shows that CloudWatch+ is both online and light-weight, and the detection results are much more detailed than those provided by AWS CloudWatch.

Acknowledgement

Research is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61472214, the National Key Basic Research Program of China (973 program) under Grant No. 2013CB329105, the State Key Program of National Science of China under Grant No. 61233007, the National Natural Science Foundation of China (NSFC) under Grant No. 61472210, the National High Technology Development Program of China (863 program) under Grant No. 2013AA013302.

We are grateful for the anonymous reviewers' thorough comments as well as the timely assistances provided by co-chairs during our submission.

References

[1] Simic Bojan. The performance of web applications. Technical report, 2008.

[2] Amazon cloudwatch. http://aws.amazon.com/cloudwatch.

[3] Windows azure monitor metrics. http://www.windowsazure.com/en-us/manage/services/web-sites/how-to-monitor-websites.

[4] Rightscale monitorsystem. http://support.rightscale.com/12-Guides/RightScale_101/08-Management_Tools/Monitoring_System.

[5] New relic monitor metrics. https://newrelic.com/docs/instrumentation/metric-types.

[6] Mohammad Hajjat, David Maltz, Sanjay Rao, Kunwadee Sripanidkulchai, et al. Dealer: application-aware request splitting for interactive cloud applications. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 157–168. ACM, 2012.

[7] V. Cardellini, E. Casalicchio, and M. Colajanni. A performance study of distributed architectures for the quality of web services. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, pages 10 pp.–, 2001.

[8] Mauro Andreolini, Michele Colajanni, Riccardo Lancellotti, and Francesca Mazzoni. Fine grain performance evaluation of e-commerce sites. *SIGMETRICS Perform. Eval. Rev.*, 32(3):14–23, December 2004.

[9] Kaiqi Xiong and H. Perros. Service performance and analysis in cloud computing. In *Services - I, 2009 World Conference on*, pages 693–700, 2009.

[10] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. Ddos-shield: Ddos-resilient scheduling to counter application layer attacks. *IEEE/ACM Transactions on Networking (TON)*, 17(1):26–39, 2009.

[11] Rewrite engine. http://httpd.apache.org/docs/2.0/misc/rewriteguide.html.

[12] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is np-hard. In *WALCOM: Algorithms and Computation*, pages 274–285. Springer, 2009.

[13] Fionn Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

[14] L. Cherkasova, K. Ozonat, Ningfang Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 452–461, 2008.

[15] Mercury diagnostics. https://www.mercury.com/us/products/diagnostics/.

[16] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. Ddos-shield: Ddos-resilient scheduling to counter application layer attacks. *IEEE/ACM Trans. Netw.*, 17(1):26–39, February 2009.

[17] N. Poggi, D. Carrera, R. Gavalda, J. Torres, and E. Ayguade. Characterization of workload and resource consumption for an online travel and booking site. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10, 2010.

[18] George EP Box, Gwilym M Jenkins, and Gregory C Reinsel. *Time series analysis: forecasting and control*. John Wiley & Sons, 2013.

[19] Yingying Chen, Ratul Mahajan, Baskar Sridharan, and Zhi-Li Zhang. A provider-side view of web search response time. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 243–254. ACM, 2013.

[20] He Yan, Ashley Flavel, Zihui Ge, Alexandre Gerber, Daniel Massey, Christos Papadopoulos, Hiren Shah, and Jennifer Yates. Argus: End-to-end service anomaly detection and localization from an isp's point of view. In *INFOCOM, 2012 Proceedings IEEE*, pages 2756–2760. IEEE, 2012.

[21] Suk-Bok Lee, Dan Pei, M Hajiaghayi, Ioannis Pefkianakis, Songwu Lu, He Yan, Zihui Ge, Jennifer Yates, and Mario Kosseifi. Threshold compression for 3g scalable monitoring. In *INFOCOM, 2012 Proceedings IEEE*, pages 1350–1358. IEEE, 2012.