RAD: Detecting Performance Anomalies in Cloud-based Web Services

Joydeep Mukherjee
York University
Email: jmukherj@yorku.ca

Alexandru Baluta
York University
Email: balutaal@yorku.ca

Marin Litoiu
York University
Email: mlitoiu@yorku.ca

Diwakar Krishnamurthy
University of Calgary
Email: dkrishna@ucalgary.ca

***Abstract*—Web services hosted on public cloud platforms are often subjected to performance anomalies. Runtime detection of such anomalies is crucial for operations in cloud data centers. With ever-increasing data center size, complexities in software applications and dynamic traffic workload patterns, automatically detecting performance anomalies is a challenging task. In this paper, we propose RAD, a lightweight runtime anomaly detection technique that does not require application level instrumentation and can be easily implemented for detecting anomalies in multi-tier cloud-based Web services. In particular, we focus on anomalies that are difficult to detect by simply monitoring system level metrics alone, such as anomalies that are caused by contention from within a service and also those caused by shared resource contention by other services running on the cloud. RAD continuously monitors service resource metrics and uses a queuing network model to detect performance anomalies at runtime. Additionally, RAD uses historical data and implements a statistical methodology to diagnose the root cause of an anomaly. We evaluate RAD on a private cloud and also on the EC2 public cloud platform to show that RAD incurs extremely low levels of performance overhead on the service and is effective for detecting anomalies in both multi-tier monolithic services and microservices.**

I. Introduction

Given the ever-increasing scale of data centers along with the increasing complexity of cloud abstraction, software architecture and dynamic workload patterns, it is important to maintain the performance of services hosted on the cloud at runtime without the need for manual intervention. Services running inside the cloud are often prone to performance anomalies due to reasons such as software bottlenecks, shared resource contention, and hardware failures [1], [2], [3], [4].

A performance anomaly is a deviation from a predefined performance profile and can manifest as an unexpectedly high request response time or/and a reduced request throughput. Anomaly causes can be broadly divided into two categories: **internal** and **external**. Internal causes include software bottlenecks such as unavailability of software resources and bugs in application code. They can also include hardware bottlenecks such as CPU hogging and resource capacity saturation. External causes, also known as **performance interference**, include resource contention by services belonging to other cloud subscribers that are competing for hardware or software resources and can lead to huge performance degradation [5] in Web services. Automatically detecting these anomalies at runtime and their causes is crucial for maintaining the performance of these services.

Detecting internal and external anomalies is challenging. Cloud providers typically do not offer built-in mechanisms to continuously monitor and detect different kinds of performance anomalies at runtime. Consequently, cloud subscribers need to deploy their own mechanisms to detect and diagnose performance issues. This is difficult since cloud subscribers do not have access to host physical machine (PM) level metrics and thereby can not use hardware counters to detect anomaly as done in the past [6], [7], [8].

Anomaly detection techniques [9], [10] proposed earlier have focused on instrumenting performance metrics such as request response times. However, such application-level instrumentation for a complex multi-tier service can be difficult to implement and can significantly increase the cost of development and maintenance. Past research has addressed the problems of instrumenting application code and continuously monitoring the time taken by various tiers of the service [11]. In addition, continuous monitoring of service response times can incur a prohibitive overhead when the application is facing a heavy workload [12], [13]. Finally, monitoring only the request response times of a service is not enough to diagnose why the anomaly occurs. This is important to understand since anomalies that are caused by software bottlenecks and internal bugs in the application code cannot be mitigated by simply scaling the service. This necessitates techniques that do not require extensive service level instrumentation, have low performance overhead and enable diagnosing the root cause for the anomaly.

In this paper, we present a novel **R**untime **A**nomaly **D**etection (**RAD**) approach for cloud-based multi-tier Web services. RAD implements the idea of a *twin workload*, which is a representative sample of the service workload, and is used to infer whether the service is currently experiencing anomalous behaviour. The twin workload is continuously submitted to the service at runtime in a controlled manner so as to result in minimal performance overhead. RAD monitors the twin's request response times along with system metrics that can be easily collected from inside the service virtual machines (VMs). Using a Queuing Network Model for the twin workload, RAD compares the runtime response time of the twin to its model predicted baseline response time in absence of an

2159-6190/20/$31.00 ©2020 IEEE
DOI 10.1109/CLOUD49709.2020.00073

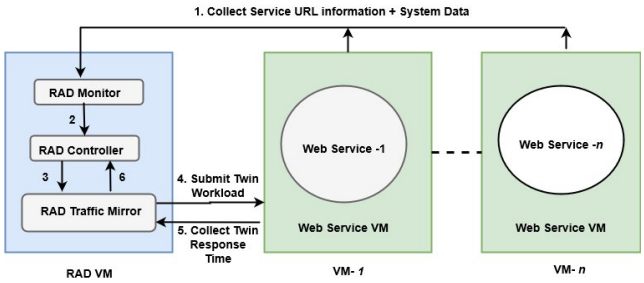

Fig. 1. Overview of RAD

anomaly. RAD indicates the presence of a performance anomaly if there is a statistically significant deviation between these two. Once an anomaly has been detected, RAD finds the root cause of the anomaly. In case of an internal anomaly, RAD isolates the software component or the service tier or the request URL that triggers the anomaly using historical data collected from the system.

We validate RAD on a private cloud setup as well as on the Amazon EC2 public cloud platform. We first validate RAD against anomalies induced by internal causes, specifically software resource saturation bottlenecks [4] caused by improper software configuration as well as internal application bugs such as a slow database queries [11], [14]. We also validate RAD against anomalies due to external causes, for e.g., performance interference from other services in a public cloud platform. We use two well-known Web service benchmarks for validating RAD: a monolithic multi-tier Web application called RUBiS [15] and a Web microservice called Acme Air [16]. RAD outperforms existing anomaly detection techniques and successfully detects internal and external anomalies while incurring extremely low levels of performance overhead on the service.

II. RAD

In this section, we present an overview of the RAD technique. Figure 1 shows the high level details of RAD. As seen in the figure, RAD is implemented inside a RAD VM that is hosted on the same cloud platform as the Web service VMs. The n Web services of the application are hosted on VM-1 to VM-n. The RAD VM hosts the three main RAD components, namely the RAD monitor, the RAD controller and the RAD traffic mirror. Next, we describe the workings of these components in more detail.

A. RAD Monitor

The RAD monitor is in charge of continuously monitoring the service VMs, collecting service and system level data from inside the VMs (step 1 in Fig. 1) and sending this data to the RAD controller (step 2 in Fig. 1). Specifically, the RAD monitor collects the following data for each sampling interval: URLs of incoming requests to the service, resource utilization values from each of the n VMs hosting the n-tiered service, and the software parallelism level in each VM. The software parallelism level is service dependent and refers to the number of active threads, processes, containers or connections in a software pool at each service level. In our case, the RAD monitor collects the number of active software threads from the top P processes running in each VM as the software parallelism level. The RAD monitor aggregates the data and passes this information to the RAD controller over a sampling interval. We choose the sampling period in a way such that the RAD monitor incurs only a small performance overhead on the service.

The RAD controller uses the data collected from the RAD monitor over a sampling interval in the following manner. First, the RAD controller instructs the RAD traffic mirror to submit the twin workload (step 3 in Fig. 1) to the service (step 4 in Fig. 1) and collects its request response times (step 5 in Fig. 1). This data is next passed to the RAD controller (step 6 in Fig. 1) which then uses this information for anomaly detection decisions, as described in Section II-C.

B. RAD Traffic Mirror

The RAD traffic mirror allows us to estimate the response time of the service at runtime without instrumenting the service directly. To this end, the RAD traffic mirror implements the concept of a twin workload, which is a representative collection of the service URLs. The twin workload is constructed once in a pre-deployment stage and is continuously submitted to the service at runtime. Note that the RAD traffic mirror substitutes all user information in the twin workload with a RAD user that is specially created for this purpose. Doing so allows for user anonymity and does not require transaction rollbacks. We follow the tuning process suggested in past work [17] to adjust the workload intensity of the twin workload such that it imposes very little overhead on the service being monitored.

C. RAD Controller

The RAD controller aggregates and analyses data from the RAD monitor and the RAD traffic mirror to detect and diagnose performance anomalies at runtime. To this end, the RAD controller first constructs a QNM in an offline model construction phase prior to deploying the service. The QNM represents a simple generic queuing model that does not take into account performance anomalies such as software bottlenecks and interference. Next, the controller uses the constructed QNM at runtime for predicting the request response times of the twin workload. Since the QNM captures only hardware contention in the system, if the model predicted response time of the twin shows statistically higher deviation than the measured runtime response time of the twin, we infer the presence of performance anomalies such as software bottlenecks or interference.

The RAD controller continuously monitors and archives system data from the RAD monitor as well as the runtime and QNM predicted response times of the twin workload at each interval for making anomaly detection decisions and diagnosing its root cause. In a given sampling interval, the anomaly detection algorithm first compares the QNM predicted baseline response time of the twin against its runtime response time to infer the presence of anomalies, if any. If an anomaly is detected, the algorithm next proceeds to isolate its root cause. To this end, the algorithm looks at historical data collected at a similar service workload mix in the absence of anomalies in the system. A service workload mix is defined by the number and type of requests belonging to each user connection and the arrival rate of user connections into the system. From this historical data, the controller then constructs software parallelism and twin workload URL response time groups and compares them against the corresponding runtime software parallelism levels and twin URL response times. Using standard statistical outlier detection techniques, the algorithm isolates the software component or the URL that triggers the anomaly. The details of the queuing network model and the anomaly detection algorithm are discussed next in Sections II-C1 and II-C2, respectively.

1) Queuing Network Model: QNMs model a system as a network of queues where each resource in the system represents a queue. We implement our QNM as a multi-class, product-form, model with two classes of workloads. Class a represents the service workload whereas class t represents the twin workload. The twin workload class t is modeled as an open class with arrival rate λ_t connections per second (cps). As mentioned previously, the n-tiered service is hosted on n cloud instances or VMs with each tier in its own VM. Each VM m is assumed to have K resources in total. The inputs to the QNM are the arrival rate of class t, i.e. λ_t, and the total utilization U_k of each resource k in m. The output of the QNM predicts the baseline response time of the twin workload over each sampling interval at runtime.

For using the QNM at runtime, the RAD controller has to first estimate the resource demands of the twin workload. This needs to be done only once in an offline model construction phase prior to service deployment. In this phase, the twin workload is submitted to the service at a known arrival rate λ_t over the duration of a sampling interval. eq. 1 estimates the service demand $D_{t,k}^{\hat{m}}$ of a resource k caused by a twin workload class t in VM m. The average residence time $R_{t,k}^{\hat{m}}$, i.e., the total time spent by a connection belonging to class t at resource k in VM m, is estimated as shown in eq. 2. Finally, the response time $\hat{R}_t$ of a connection belonging to class t is estimated using eq. 3, which is the sum of residence times of the connection at all K resources in VM m taken over n VM tiers. The controller also obtains the average response time R_t of the twin workload in the interval during the offline model construction phase.

$$D_{t,k}^{\hat{m}} = \frac{U_{t,k}^{m}}{\lambda_t} \tag{1}$$

$$R_{t,k}^{\hat{m}} = \frac{D_{t,k}^{\hat{m}}}{1 - U_k} \tag{2}$$

$$\hat{R}_t = \sum_{m=1}^{n} \sum_{k=1}^{K} R_{t,k}^{\hat{m}} \tag{3}$$

Next, the RAD controller revises the estimated service demand $D_{t,k}^{\hat{m}}$ of the twin workload to match with its real service demand. In our experiments, we found that the observed values of R_t for the twin workload is always higher than the estimated values of $\hat{R}_t$. In light of this observation, we propose a calibration of $D_{t,k}^{\hat{m}}$ based on the estimated and observed values of $\hat{R}_t$ and R_t as given by eq. 4

$$D_{t,k}^{\hat{m},r} = D_{t,k}^{m} \times \frac{R_t}{\hat{R}_t} \tag{4}$$

Next, the RAD controller uses the estimated revised demand $D_{t,k}^{\hat{m},r}$ of the twin workload for predicting the baseline average connection response time $\hat{R}_t$ of the twin workload at runtime. To this end, the controller substitutes $D_{t,k}^{\hat{m}}$ with the new revised estimates of $D_{t,k}^{\hat{m},r}$ in eq. 2 and eq. 3 to predict $\hat{R}_t$. As shown later in Section IV-A, we validate that the predicted runtime response times of the twin workload using the revised demand estimates closely matches the twin's monitored runtime response times in the absence of anomalies.

2) Anomaly Detection Algorithm: The RAD controller employs an anomaly detection algorithm to detect software performance anomalies at runtime. For a sampling interval i, the response times of each of the S URLs of the twin workload at second m in sampling interval i is represented as a tuple $R^{i,m}$ as shown in eq. 5. The number of active software threads from the top P processes in each VM for all n VMs in the system at second m in i is represented as a group of the software parallelism level in a tuple $C^{i,m}$ as shown in eq. 6.

$$R^{i,m} =< R_1^{i,m}, R_2^{i,m} ... R_S^{i,m} > \tag{5}$$

$$C^{i,m} =< C_1^{i,m}, C_2^{i,m} ... C_N^{i,m} > \tag{6}$$

where, $N = n \times P$.

We now describe the RAD algorithm in detail as shown in Algorithm 1. The algorithm calculates the 95% confidence interval (CI) of the average response time of all S URLs in the twin at each second m in an interval i, along with the twin's non-anomalous QNM predicted baseline average response time $\hat{R}_t$. Next, it uses standard statistical outlier detection techniques to see if the runtime response time of the twin is statistically higher than $\hat{R}_t$. If so, an

Algorithm 1 RAD algorithm

for $<$ *each sampling interval* $i>$ **do**
  $R^{i,m}_{avg} \leftarrow \sum_{x=1}^{S} R^{i,m}_{x}/S$
  $R^{i}_{avg} \leftarrow \sum_{m=1}^{T} R^{i,m}_{avg}/T$
  calculate 95% *CI of* $R^{i,m}_{avg}$ *in* i
  calculate $\hat{R}_t$ *in* i *using QNM*
  if $((R^{i}_{avg} > \hat{R}_t) \& \hat{R}_t \notin (95\%\ CI\ of\ R^{i,m}_{avg}))$ **then**
    find j *intervals from history with same mix where* $\hat{R}_t \in (95\%\ CI\ of\ R^{i,m}_{avg})$
    do
    $C_{normal} = \{< C^1_1..C^j_1 >, .. < C^1_N..C^j_N >\}$
    $R_{normal} = \{< R^1_1..R^j_1 >, .. < R^1_S..R^j_S >\}$
    done
    for seconds in i *where* $R^{i,m}_{avg} > \hat{R}_t$
    do
    $C_{anomaly} = \{C_{1,an}, C_{2,an}...C_{N,an}\}$
    $R_{anomaly} = \{R_{1,an}, R_{2,an}...R_{S,an}\}$
    done
    for $< r = 1$ *to* $N >$ **do**
      if $C_{r,an}$ *in* $C_{anomaly}$ *is an outlier w.r.t. the* r*th tuple in* C_{normal} **then**
        Flag Anomaly
        Flag VM and process corresponding to r *as root cause of anomaly*
        for $< k = 1$ *to* $S >$ **do**
          if $R_{k,an}$ *in* $R_{anomaly}$ *is outlier w.r.t* k*th tuple in* R_{normal} **then**
            Flag URL corresponding to $R_{k,an}$ *as root cause*
          end if
        end for
      end if
      else
      Run probe for detecting interference
    end for
  end if
end for

anomaly is detected in the system and the RAD algorithm proceeds to infer the root cause of the anomaly.

To infer the root cause, the algorithm first looks at historical data to find j non-anomalous sampling intervals with the same service workload mix. From these intervals, the algorithm groups past software parallelism levels and URL response times obtained from the system in absence of anomaly in lists C_{normal} and R_{normal}, respectively. Next, the RAD algorithm groups corresponding anomalous system data in lists $C_{anomaly}$ and $R_{anomaly}$ from the current sampling interval i. The algorithm then compares each element in $C_{anomaly}$ with its corresponding element in C_{normal}. If the rth element in $C_{anomaly}$ is identified as a statistical outlier in comparison to the rth element in C_{normal}, the algorithm flags the VM and the software process corresponding to the rth element in $C_{anomaly}$ as the probable root cause related to the anomaly. As part of the root cause analysis, the RAD approach also investigates if the root cause is associated with increased response times of specific URL(s). To this end, the algorithm compares each element in $R_{anomaly}$ with its corresponding element in R_{normal}. If the kth element in $R_{anomaly}$ is identified as a statistical outlier in comparison to the kth element in R_{normal}, the algorithm infers that the anomaly impacts the response time of the kth URL in the service workload mix.

If no internal anomalies are detected, the RAD algorithm next checks to see if the root cause of the anomaly is caused by an external bottleneck, i.e., performance interference. For this purpose, RAD utilizes past research [17] by running a low overhead software probe in each service tier along with the service for the period of a sampling interval. The probe represents a Web micro-benchmark that is designed to utilize the system resources at each tier. By subjecting the probe micro-benchmark to a controlled workload, RAD imposes minimal overhead on the service response time. RAD first estimates the baseline no-interference response times of the probe at various levels of resource utilization by using a dedicated instance in an offline phase prior to service deployment, similar to Section II-C1. Next, using the same statistical outlier detection technique as before, RAD compares the runtime response time of the probe against its estimated baseline response time for detecting the presence of an external anomaly.

III. Experimental Setup and Anomalies

A. Private Cloud Setup

The private cloud setup consists of a dual socket Intel Xeon E5645 server host with 6 cores per socket. Multiple VM instances are consolidated on this server where each instance is configured with 1 virtual CPU (VCPU) and 1 GB of physical memory. We consider a 2-tier Web service for our private cloud experiment setup. As mentioned earlier, each tier of the service is hosted inside an instance. To this end, we spin 2 instances on 2 sockets of the server with each instance pinned to a socket. We use another identical server to host the RAD VM. The RAD monitor inside the RAD VM monitors the resource utilization metrics every second for the following resources in each service instance: VCPU, memory, disk, network. The RAD monitor also collects data regarding the service workload mix by instrumenting the service Web server and recording the URL information received by the service in a sampling interval. Note that this is easy to do since it involves instrumentation at only the instance hosting the Web server tier for the service. In our case, we activate Web server logging in the Web tier instance to record the URL information for our service. Finally, the RAD monitor collects the number of active software threads from the top 10 processes in each instance. We note that the RAD

monitor does not impose any considerable overhead either on the service or on the RAD VM.

The RAD traffic mirror runs the *httperf* [18] workload generator tool inside the RAD VM to submit the twin workload. The twin workload is submitted to the system by the RAD traffic mirror using *httperf* at the rate of C HTTP connections per second (cps) with an exponentially distributed inter-arrival time over the sampling interval of T seconds. We configured the workload twin with $C = 2$ cps and $T = 10$ seconds since we observed that these values for the workload twin incur a maximum of $2-3\%$ response time overhead on the service.

B. EC2 Setup

We use the EC2 setup to validate RAD with a Web microservice benchmark in detecting performance anomalies such as performance interference, which is a common anomaly in public cloud platforms. To this end, we use 3 m4.large instances in EC2. Similar to our private setup, we consider a Web microservice benchmark with 2 tiers or microservices for the EC2 setup. Each microservice is configured as a Docker container in a m4.large EC2 instance. We host the RAD VM in a separate m4.large instance. The configurations for the RAD monitor, the RAD traffic mirror and the RAD controller are the same as that used in Sec. III-A before. We confirm that these settings impose a negligible maximum overhead of $2-3\%$ on the service's request response time.

As described in Section II-C2, RAD runs a probe micro-benchmark in each service tier over the short duration of one sampling interval to confirm the presence of interference. To this end, the probe runs a Docker container in each instance executing a synthetic micro-benchmark which incurs exponentially distributed resource demands. We verify that running the probe container along with our microservice containers incurs a maximum overhead of only $3-4\%$ on the service's response time.

C. Web Service Benchmarks

We consider two different Web services for evaluating the RAD approach. The first Web service is a 2-tier Web benchmark called RUBiS. The 2 tiers in RUBiS are called the Web tier and the database tier respectively and are hosted on 2 separate VMs running on our private cloud. We use the default *browsing* transaction workload mix specified by RUBiS which emulates 5000 users issuing a series of 14 inter-dependent requests to the Web tier VM. We also configure a *database* transaction mix for RUBiS where a majority of user requests query the MySQL RUBiS database in the database tier VM.

The second service is an open source Web microservice benchmark called Acme Air that emulates transactions for an airline website. We use 2 microservices to host the Acme Air service inside 2 EC2 instances. Both these microservices are run inside Docker containers on the instances. The workload transaction mix submitted to Acme Air is the default workload obtained from the official Acme Air project [19].

D. Performance Anomalies

1) Software Configuration Anomaly: We now discuss the performance anomalies we consider. We inject a software bottleneck anomaly within the RUBiS benchmark in our private cloud setup that occurs due to improper configuration at the software level [20], [4]. We configure the maximum number of Apache threads to 40 in the Web tier VM, which implies that any user request that arrives at the Web tier VM when there are already 40 or more concurrent requests will get queued at the software level waiting for additional Apache threads to be spawned. Consequently, the average request response time will increase since some of the incoming requests will have to wait for a longer time due to unavailability of software threads.

To demonstrate the impact of this anomaly, we conduct an experiment where we run 2 tests. Both tests submit the browsing workload mix to the RUBiS benchmark at the same connection arrival rate. However, in the second test, the **persistent connections** setting in Apache is enabled which causes a higher number of concurrent Apache threads in the Web tier VM. Once the number of concurrent connections at the Web tier is higher than the maximum limit of Apache threads configured, a software bottleneck anomaly is created. The results of the experiment are shown in Table I. As seen in the table, the average response time of a request in the RUBiS browsing mix is significantly higher when the persistent connection configuration is set to **On**.

From our experiment results, we note that detecting a software configuration anomaly is challenging. As seen in Table I, the arrival rates in both cases are configured to the same value of 10 cps. Consequently, the average CPU utilization values at the Web tier VM with and without enabling persistent connections are very similar to each other. Other hardware resource utilization values collected at both tiers also do not show any significant change in both cases. Simply monitoring the incoming user arrival rate and the resource utilization metrics in the system will not indicate the presence of this anomaly. This show the need for an intelligent model-driven technique such as RAD.

2) Query Latency Anomaly: The second software bottleneck anomaly we consider is an anomaly caused by a latency in a database query as reported in previous work [11], [14]. We inject a sleep delay of 1 second in the *SearchItemsbyCategory.php* script that invokes a MySQL query to the database tier VM. As a result, a MySQL query initiated by this script takes a long time to complete and causes subsequent MySQL queries to queue in the database tier.

We conduct two sets of tests in our private cloud setup using the database workload mix configured with and without the latency delay in the *SearchItemsbyCate-*

TABLE I
Software Configuration Anomaly

Workload Mix	Arrival Rate (cps)	Persistent Connection	Response time (ms)	CPU Utilization(%)	Concurrent Requests
browsing	10	**Off**	5.8	17.8	19
browsing	10	**On**	31.9	18.4	115

TABLE II
Query Latency Anomaly

Workload Mix	Arrival Rate (cps)	Query Latency	Response time (ms)	Web tier CPU U(%)	DB tier CPU U(%)	Apache Threads	MySQL Threads
database	10	**Off**	14.2	36	67	751	71
database	10	**On**	133.7	28	41	840	384

gory.php script. Table II shows the results of our experiments. As seen in the table, the average request response time of the database workload mix in RUBiS increases by 40% when a query latency delay is introduced. Similar to the software configuration anomaly, the query latency anomaly is challenging to detect since it manifests at the same arrival rate for the same database workload mix. Moreover, as seen in the table, the average CPU utilization in the Web and database tiers decrease when this anomaly manifests itself, thereby incorrectly indicating the absence of a performance problem.

3) Performance Interference Anomaly: Performance interference is an example of an external bottleneck that can happen on public cloud platforms [17], [5], [21]. We motivate the performance interference anomaly in the Acme Air microservice benchmark on our EC2 setup. We introduce performance interference by running a copy of the front-end and back-end service containers on the same EC2 instances as the original Acme Air benchmark. The incoming request arrival rate to the Acme Air containers is fixed at 100 cps. We vary the workload to the interfering containers such that they incur increasing amount of CPU contention on the instances. Figure 2 shows the effect of the performance interference anomaly on the response time of the Acme Air benchmark. As seen in the figure, the mean request response time of the Acme Air benchmark increases from 5.7 ms to 25.5 ms when the arrival rate for the interfering containers increases from 0 to 50 cps. However, the CPU utilization incurred by the Acme Air front-end and back-end containers remains unchanged at 40% and 5% respectively. This shows that the performance anomaly is difficult to detect by monitoring only the resource utilization levels of the containers.

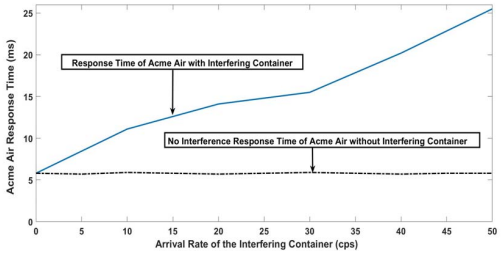

Fig. 2. Performance Interference in Acme Air

IV. Results

A. QNM Validation

We first validate the QNM used by RAD on our private cloud setup. As mentioned previously, the RAD controller uses the QNM to predict the baseline response time $\hat{R}_t$ of the twin workload at runtime. We run an experiment where the QNM first uses eq. 1, eq. 2 and eq. 3 to estimate $\hat{R}_t$ for the twin workload. However, we We observed a maximum mismatch of 39% between the $\hat{R}_t$ and the monitored response time R_t values for the twin. We follow the QNM calibration described earlier in Section II-C1 to improve the QNM prediction. The QNM uses the values of $\hat{R}_t$ and R_t as inputs to eq. 4 to estimate the revised resource demands for the twin workloads. Next, we conduct 10 experiments where we estimate $\hat{R}_t$ using the revised resource demand estimates at different workload arrival rates. Table III shows a subset of the experiment results. As seen in the table, the $\hat{R}_t$ and R_t values of the twin workload are very close to each other with a maximum prediction mismatch of around 4% across all experiments. The QNM is similarly validated in EC2 with a maximum prediction mismatch of around 5% across all experiments.

B. RAD in Private Cloud

We now conduct experiments to validate RAD against the software configuration anomaly and the query latency anomaly on our private cloud. We conduct two experiments, one for each anomaly, with each experiment set to a duration of 200 seconds. The RAD sampling interval is set to 10 seconds.

We first run an experiment where we introduce the software configuration anomaly in RUBiS at the 100 second

TABLE III
QNM Validation in Private Cloud

Arrival Rate (cps)	$\hat{R}_t$ (ms)	R_t (ms)
10	5.8	5.9
20	10.6	10.8
30	16.9	17.6
40	22.3	23.7
50	28.7	29.9

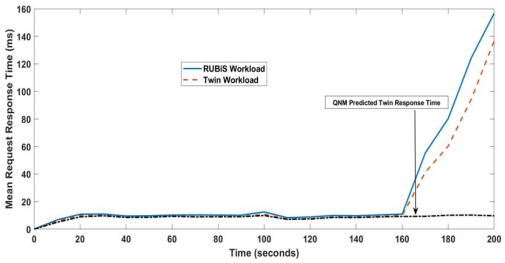

Fig. 3. RAD With Software Configuration Anomaly

mark and validate RAD against it. Figure 3 shows the results of our experiment. As seen in the figure, the mean request response time of the RUBiS workload remains steady at around 9.8 ms for the first 160 seconds and then registers a steady increase up to a maximum value of 156.8 ms due to the software configuration anomaly. Consequently, the monitored mean request response time of the twin workload also closely matches the response time pattern of the RUBiS workload. However, the QNM predicted baseline response time of the twin remains steady at around 9.6 ms for the entire duration of the experiment. As a result, RAD correctly indicates the presence of a performance anomaly in all sampling intervals starting from 160 seconds.

In addition to correctly predicting the presence of the software configuration anomaly, RAD also correctly points to the root cause of this anomaly. In our experiment, the RAD controller infers that the *apache2* thread counts in the Web tier VM when the anomaly is present is an outlier when compared to the *apache2* thread counts collected from previous non-anomaly sampling intervals. Consequently, RAD correctly indicates the Web tier VM and the Apache2 Web server application as the root source of the anomaly.

Next, we validate RAD at runtime to see if it can detect the query latency anomaly in RUBiS. For this purpose, we design an experiment where we introduce a query latency delay in RUBiS at the 100 seconds mark. Figure 4 shows the results of our experiment. As seen in the figure, the response time of the RUBiS workload remains steady at around 16.5 ms for the first 100 seconds and then increases steadily to a maximum of 169.1 ms after that. Consequently, the monitored response time of

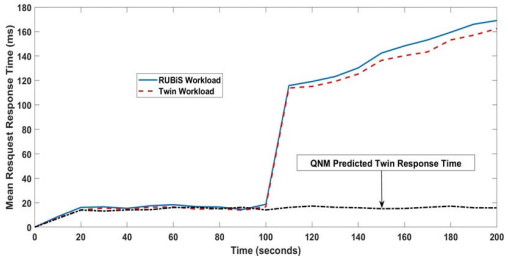

Fig. 4. RAD With Query Latency Anomaly

the twin workload closely follows the response time of RUBiS. Using the predicted baseline response time of the twin workload, RAD successfully detects the presence of a performance anomaly in all sampling intervals beginning from the 100 seconds mark. Next, RAD uses the software parallelism level data in these intervals and isolates the database tier as the root cause of the anomaly since it shows a statistically high level of parallelism as compared to past intervals with non-anomalous behaviour. Finally, RAD compares the anomalous URL response times with historical data and correctly indicates the *SearchItemsby-Category.php* script as the impacted URL.

C. RAD in EC2

We conduct experiments to demonstrate the validity of RAD on our EC2 setup. To this end, we test RAD against the performance interference anomaly. We conduct an experiment such that after 100 seconds from the beginning of the experiment, we inject the performance interference anomaly in Acme Air for the next 100 seconds. Figure 5 shows the results of our experiment. As seen in the figure, the response time of Acme Air remains steady at around 5.8 ms for the first 100 seconds and then increases to a maximum of 16.4 ms for the next 100 seconds. Since the monitored response time of the twin during the last 100 seconds is substantially higher than its QNM predicted baseline response time, RAD first checks to see if the anomaly is caused by an internal anomaly by comparing the runtime system data against historical data incurred by a similar service workload. Since there is no statistical difference between these 2 sets of data, RAD next runs the probe on both instances and compares the runtime response time of the probe to its estimated baseline response time. The probe response times show a statistically significant increase, thus confirming the presence of interference in the system.

D. Comparison with baseline methods

Past anomaly detection and management techniques [21] have monitored hardware performance counters such as cycles per instruction and cache miss rate in cloud instances. However, public cloud subscribers can not use these techniques since subscribers do not get access to hypervisor or host level hardware counters on public

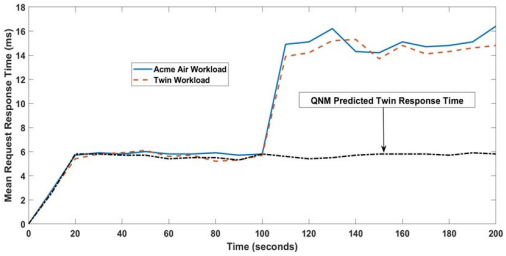

Fig. 5. RAD With Performance Interference Anomaly

cloud platforms. We verified this behaviour in EC2 where we were not given access to monitor hardware counters for our VMs. In the absence of hardware counters, we validate RAD against threshold-based anomaly detection techniques, as proposed in prior work [22]. To this end, we investigate how statically set threshold values on CPU utilization compare against our RAD technique. We observe that the average CPU utilization of the VMs in our private cloud in the sampling intervals before and after the software configuration anomaly is very similar, varying between 34.1% to 36.3%. For the query latency anomaly, the CPU utilization values recorded in the sampling intervals decreased from 52% to 35%. In EC2, we use the *docker stats* utility to monitor the CPU utilization of our Acme containers. The monitored CPU utilization values for Acme in EC2 before and after the interference anomaly stayed at around the same values of 42%. Our observations in both private cloud and EC2 indicate that threshold-based detection techniques will be unable to infer the presence of anomalies caused by software bottlenecks and interference. In contrast, our model-based RAD technique successfully detects these anomalies.

V. Sensitivity Analysis

In this section, we run experiments to validate RAD against new software configurations such as an asynchronous Web server in the RUBiS benchmark and an internal bottleneck such as a query latency anomaly in the Acme Air benchmark. To this end we use the *lighttpd* asynchronous Web server to host RUBiS. We set up 2 experiments on our private cloud where we configure lighttpd with 1 and 4 processes respectively. We submit the browsing workload to RUBiS at an arrival rate of 10 cps for 100 seconds in both cases. Table IV shows the results of our experiment. As seen in the table, the response time of RUBiS almost doubles when the number of lighttpd processes in the system increases from 1 to 4. Note that this is a performance anomaly since the average CPU utilization in the Web tier during both experiments is similar. We validate that our RAD approach can correctly detect the anomaly and its root cause.

Next, we conduct an experiment to show how RAD can detect the query latency anomaly in Acme Air. To this end, we run a 200 second experiment where we

TABLE IV
Asynchronous Web Server with RAD

Lighttpd Processes	CPU Utilization (%)	Response time (ms)
1	44.7	18.9
4	47.3	34.3

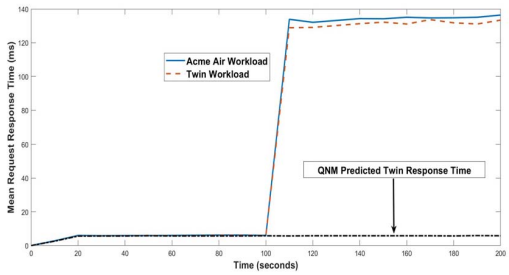

Fig. 6. RAD With Query Latency Anomaly in Acme Air

introduce an asynchronous query latency of 1000 ms in the *queryflights* URL in Acme Air during the last 100 seconds of the experiment. As seen in Figure 6, the mean request response time of Acme Air increases from around 6 ms to 133 ms due to the query latency anomaly. Our RAD approach is able to successfully detect the query latency anomaly and isolate the *queryflights* URL as the impacted URL.

VI. Related Work

Past work has looked into detecting and predicting performance anomalies caused by hardware bottlenecks. For example, Tan *et al.* propose an automated performance anomaly prevention system called PREPARE [23] that can predict performance anomalies caused by faults such as memory leaks, CPU hogs and resource capacity bottlenecks. Cherkasova *et al.* present an anomaly detection framework [20] using regression based transaction models and application signatures to detect anomalies caused by hardware resource consumption. NAP [24] collects network communication traces and uses a queuing theory based model to detect anomalies caused by hardware resource overloading. Similarly, PAL [25] proposes a non-intrusive anomaly localization system that detects anomalies caused by events such as memory leaks and CPU hogging. In contrast to these work, RAD specifically looks at performance anomalies caused by software bottlenecks and performance interference, which are difficult to detect since they do not manifest as changes in hardware resource utilization patterns.

Real time anomaly detection attempts that focus on software bottlenecks have been reported in several papers. For example, Jayathilaka *et al.* present a real time monitoring and diagnostic framework called Roots [11] that can identify root cause of performance anomalies

in Platform-as-a-Service (PaaS) clouds by using a combination of metadata injection and platform-level instrumentation. Mukherjee *et al.* proposes a cloud provider oriented interference detection technique [26] that involves running a probe directly on the PM in a cloud platform. Similar interference-aware techniques were proposed by others [27], [8] for Web services in public cloud platforms. In contrast, RAD can be used by cloud subscribers since it does not need access to underlying platform and PM-level metrics.

VII. Conclusions and Future Work

In this work, we propose a novel technique called RAD that can be used by cloud subscribers to detect performance anomalies in cloud-based Web services at runtime. In contrast to past work, we focus exclusively on anomalies caused by software bottlenecks and interference that are hard to capture using existing techniques. RAD executes a twin workload and uses the QNM predicted baseline response time of the twin to infer anomaly. RAD further investigates the root cause of the anomaly such as isolating the service tier where the anomaly manifests itself and the URL responsible for the anomaly. We validate RAD against frequent anomaly types caused by internal bottlenecks such as incorrect software configuration and query latency bottlenecks and external interference. Our test results show that RAD introduces negligible levels of overhead and is effective in detecting anomalies in both private and AWS EC2 public cloud against monolithic Web services as well as container-based microservices.

In the future, RAD can be extended to include other commonly found performance anomalies. Specifically, we will look at performance anomalies that are caused by other types of software bottlenecks in a system. We will validate the accuracy of RAD by reporting the false positive and negative rates as obtained from experiments done with a larger set of anomalies. RAD can be extended to cover scenarios where the service is dynamic in nature, i.e. where the service tiers can scale out and scale in based on the incoming traffic. Future work will also focus on evaluating RAD on a larger scale, using a larger number of cloud instances and microservices.

References

[1] S. Pertet and P. Narasimhan, "Causes of failure in web applications," Technical Report CMU-PDL-05-109, Carnegie Mellon University, Tech. Rep., 2005.

[2] J. P. Magalhaes and L. M. Silva, "Detection of performance anomalies in web-based applications," in *2010 Ninth IEEE International Symposium on Network Computing and Applications*, 2010.

[3] K. Das, "Detecting patterns of anomalies." Carnegie Mellon University, 2009.

[4] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys (CSUR)*, 2015.

[5] J. Mukherjee, M. Wang, and D. Krishnamurthy, "Performance testing web applications on the cloud," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*.

[6] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang, "Hardware counter driven on-the-fly request signatures," ser. ASPLOS XIII, 2008.

[7] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014.

[8] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[9] F. Willnecker, A. Brunnert, W. Gottesheim, and H. Krcmar, "Using dynatrace monitoring data for generating performance models of java ee applications," in *ACM/SPEC ICPE*, 2015.

[10] N. Relic, "Application performance management & monitoring," 2016. [Online]. Available: https://newrelic.com

[11] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web*, 2017.

[12] V. Horkỳ, J. Kotrč, P. Libič, and P. Tma, "Analysis of overhead in dynamic java performance monitoring," in *ACM/SPEC ICPE*, 2016.

[13] S. Agarwala, Y. Chen, D. Milojicic, and K. Schwan, "Qmon: Qos-and utility-aware monitoring in enterprise systems," in *2006 IEEE International Conference on Autonomic Computing*.

[14] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "Root cause analysis of anomalies of multitier services in public clouds," *IEEE/ACM Transactions on Networking*, pp. 1646–1659.

[15] "Rubis: rice university bidding system," 2020. [Online]. Available: https://github.com/uillianluiz/RUBiS

[16] "Acme air," 2019. [Online]. Available: https://github.com/acmeair

[17] J. Mukherjee, D. Krishnamurthy, and M. Wang, "Subscriber-driven interference detection for cloud-based web services," *IEEE Transactions on Network and Service Management*, 2017.

[18] D. Mosberger and T. Jin, "httperfâa tool for measuring web server performance," *ACM SIGMETRICS Performance Evaluation Review*, 1998.

[19] "Acme air workload driver," 2019. [Online]. Available: https://github.com/acmeair/acmeair-driver

[20] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Automated anomaly detection and performance modeling of enterprise applications," *ACM TOCS*, 2009.

[21] A. K. Maji, S. Mitra, and S. Bagchi, "Ice: An integrated configuration engine for interference mitigation in cloud services," in *2015 IEEE ICAC*.

[22] Chengwei Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online detection of utility cloud anomalies using metric distributions," in *IEEE NOMS 2010*.

[23] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*.

[24] M. Ben-Yehuda, D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg, "Nap: a building block for remediating performance bottlenecks via black box network analysis," in *ACM ICAC*, 2009.

[25] H. Nguyen, Y. Tan, and X. Gu, "Pal: P ropagation-aware a nomaly l ocalization for cloud hosted distributed applications," in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*. ACM, 2011.

[26] J. Mukherjee, D. Krishnamurthy, J. Rolia, and C. Hyser, "Resource contention detection and management for consolidated workloads," in *IM 2013*, 2013.

[27] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013.