Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications

Hiranya Jayathilaka, Chandra Krintz, Rich Wolski
Computer Science Department
University of California, Santa Barbara
{hiranya,ckrintz,rich}@cs.ucsb.edu

ABSTRACT

In this paper, we describe Roots – a system for automatically identifying the "root cause" of performance anomalies in web applications deployed in Platform-as-a-Service (PaaS) clouds. Roots does not require application-level instrumentation. Instead, it tracks events within the PaaS cloud that are triggered by application requests using a combination of metadata injection and platform-level instrumentation.

We describe the extensible architecture of Roots, a prototype implementation of the system, and a statistical methodology for performance anomaly detection and diagnosis. We evaluate the efficacy of Roots using a set of PaaS-hosted web applications, and detail the performance overhead and scalability of the implementation.

Keywords

web services; application performance monitoring; platform-as-a-service; cloud computing; root cause analysis

1. INTRODUCTION

Over the last decade cloud computing has become a popular approach for deploying applications at scale [2, 32]. This widespread adoption of cloud computing, particularly for deploying web applications, is facilitated by ever-deepening software abstractions. While abstractions elide much of the complexity necessary to enable scale, they also obscure the runtime details, making the diagnosis of performance problems challenging. Therefore, the rapid expansion of cloud technologies combined with their increasing opacity has intensified the need for new techniques to monitor applications deployed in cloud platforms [8].

Application developers and cloud administrators wish to monitor application performance, detect anomalies, and identify performance bottlenecks. To obtain this level of operational insight in a cloud, the cloud platform must support data gathering and analysis capabilities that span the entire software stack. However, most cloud technologies available today do not provide adequate means to monitor applications or the cloud services they depend on, in a way that facilitates identifying performance bottlenecks within the cloud platform.

Application-level instrumentation packages are plentiful [29, 9, 11], but because they perturb the applications they can significantly increase the effort and financial cost of application development and maintenance. Moreover, cloud administrators must trust the application developers to instrument their applications correctly (e.g. logging all the relevant information to diagnose a rare performance fault). Even when good application instrumentation is available, however, because applications depend so heavily on extant performance opaque cloud services (e.g. database services, in-memory caching, etc.), it is often difficult, if not impossible, to diagnose the "root cause" of a performance problem.

Further compounding the performance diagnosis problem, today's cloud platforms are large and complex [8, 17]. They are comprised of many layers, where each layer may consist of many interacting components. Therefore when a performance anomaly manifests in a user application, it is often challenging to determine the exact software component that is responsible. Facilitating this type root cause analysis requires both data collection at different layers of the cloud, and mechanisms for correlating the events recorded at different layers [28].

In this paper, we present *Roots* – a full-stack application platform monitor (APM) that is designed to be integrated into a variety of cloud Platform-as-a-Service (PaaS) technologies. PaaS clouds, in particular, provide abstractions that hide most of the details concerning application runtime [33]. They offer a set of scalable, managed cloud services that provide an assortment of functionality, which developers invoke from the application implementations. To be able to correlate application activity with cloud service events, Roots must be able to introspect the entire platform software stack. Therefore we have chosen to design it as another managed service built into the PaaS cloud. As an added benefit, by implementing Roots as an intrinsic PaaS service, it can function fully automatically in the background, without requiring instrumentation of application code. Instead, Roots intercepts and records events as the application code invokes various service implementations of the PaaS cloud, and then correlates them with specific application requests. Roots also records the latency of each application request, and of each application call to an internal cloud service implementation.

When Roots detects a performance anomaly in application request latency, it attempts to identify the root cause

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052649

of the anomaly by analyzing the previous workload data of the application, and the performance of the internal PaaS services on which the application depends. It determines if the detected anomaly was caused by a change in the application workload (e.g. a sudden spike in the number of client requests), or an internal bottleneck in the cloud platform (e.g. a slow database query). To facilitate this analysis we propose a "bottleneck identification" methodology for PaaS clouds. Our approach uses a combination of quantile analysis, change point detection and linear regression.

We test the efficacy of our approach with a working prototype of Roots using the AppScale [22] open source PaaS. Our results indicate that using information gathered from the entire cloud stack to parameterize the bottleneck identification algorithm, Roots makes remarkably accurate diagnoses. We also demonstrate that Roots does not add a significant performance overhead to the applications, and that a single Roots pod (our encapsulation abstraction for data analysis processes) can monitor tens of thousands of applications simultaneously.

Thus we summarize the contributions made by this paper as follows.

- We describe the architecture of Roots as an intrinsic PaaS service, which works automatically without depending upon application instrumentation.
- We describe a statistical methodology for determining when an application is experiencing a performance anomaly, and identifying the workload change or the cloud service that is responsible for the anomaly.
- We demonstrate the effectiveness of the approach using a working PaaS prototype and real web applications.

The rest of this paper is organized as follows. Section 2 describes the domain of PaaS clouds and discusses the fundamentals of performance monitoring. Section 3 overviews Roots and the motivation behind our design choices. Section 4 describes the implementation of our Roots prototype. Section 5 presents our experimental results. Finally, we discuss related work (Section 6) and conclude (Section 7).

2. BACKGROUND

We only consider web applications deployed in PaaS clouds. An application of this nature exposes one or more web application programming interfaces (web APIs) through which clients can interact with it. The web APIs accept HTTP/S requests sent by remote clients, and respond with machine readable responses (e.g. HTML, JSON, XML, Protocol Buffers). This type of applications tend to be highly interactive, and clients typically have strict expectations about application response time [23]. Additionally, the PaaS cloud on which an application is running may also impose constraints on the application response time for scalability reasons [27, 14]. For example Google App Engine (GAE) [13] requires that no application request takes longer than 60 seconds to execute.

PaaS-hosted web applications, like those that run on GAE, rely on various services offered by the underlying cloud platform. We refer to these services as PaaS kernel services or PaaS services. Offloading common application functionality such as data storage, caching, and user management to a set of scalable and managed PaaS services, significantly simplifies and expedites application development.

The downside of this approach is that application developers no longer have full visibility into application execution. Since most of the application functionality is provided by a set of PaaS kernel services managed by the cloud provider, the application developer does not have complete insight into application performance. If application response time becomes too slow, it is very challenging for the application developer to determine the cause of the performance bottleneck due to the opacity of the cloud platform's internal implementation.

One way to circumvent this limitation is to instrument application code, and continuously monitor the time taken by various parts of the application [29, 9, 11]. Unfortunately, this is tedious for the application developer, error prone thereby misleading those attempting to diagnose a problem, and the additional code instrumentation may slow down or alter the application's performance. In contrast, implementing data collection and analysis as a service built into the PaaS cloud allows anomaly detection and bottleneck identification to be a "curated" service that is reliably and scalably managed by the cloud platform.

2.1 PaaS Performance Anomalies

Our model is one in which the clients of a web application have engaged in a "service-level agreement" (SLA) [19] with the "owner" of the application that is hosted in a PaaS. The SLA stipulates a response-time "service-level objective" (SLO) that, if violated, constitutes a breach of the agreement. A performance anomaly, then, is an event or set of events that causes an application-level SLO to be violated. The response time SLO of an application is specified by its owner, when the application is deployed.

We refer to the process of diagnosing whether increased workload or an unusually slow software component is responsible for an anomaly, and in the latter case, the identification of the slow component as *root cause analysis*. When request latency increases and violates an SLO and the workload has not increased, we assume that there is a "bottleneck" component in the application code or PaaS runtime that is responsible. Roots performs root cause analysis in near real time so that it can alert application owners and cloud administrators of problems it detects. This allows application owners to identify the potential bottlenecks in their application implementations, and revise them accordingly. Cloud administrators can use this information to uncover the bottlenecks in the cloud platform, and take corrective action.

3. APPROACH

Our approach is based on the following observations.

- In a well-designed web application targeted for PaaS environments, most of the "work" is done by curated services intrinsic to the PaaS cloud [18].
- An application making maximal use of the PaaS services finds that its performance is defined by the performance of these services. Since the implementation and deployment details of the PaaS services are hidden from the applications, performance diagnostics must also be implemented as an intrinsic PaaS service.
- The integrity and accuracy of application performance diagnostics cannot rely on programmer-introduced application instrumentation.

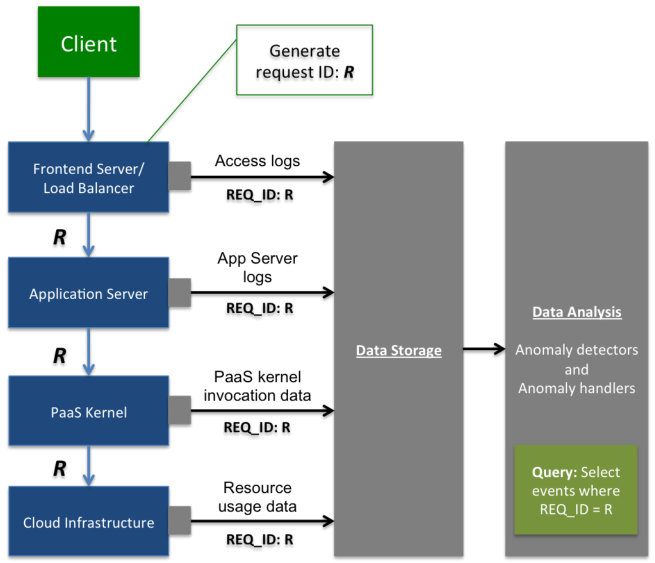

Figure 1: Roots APM architecture. Roots injects a request ID (R) at request ingress that it uses to correlate events at each level of the stack.

Thus, the key intuition behind Roots is that as a curated PaaS service it has visibility into all the activities that occur in various layers of the cloud, including all invocations of the PaaS kernel services made by the applications.

3.1 Data Collection and Correlation

Since Roots does not rely on application instrumentation, it must infer application performance traits from the performance of the PaaS services in the cloud platform. Typically, each individual layer of the cloud platform collects data regarding its own state changes facilitating measurement of the time taken in service operations. However, a layer cannot monitor state changes in other layers without violating software isolation properties of layering in general. That is, layers typically log their own internal activities, but do not (or cannot) correlate these activities with the internal activities of other layers.

Roots modifies the front-end request server (typically a software load balancer) to tag all incoming application requests with unique identifiers. Doing so enables Roots to associate invocations of PaaS kernel services with a particular request. This request identifier is attached to an HTTP request as a header, which is visible to all internal components of the PaaS cloud. We configure performance data collecting agents in each software layer in the PaaS cloud to record the request identifiers along with any events they capture. This way we maintain the relationship between application requests, and the resulting local state changes in different layers of the cloud without breaking the layered abstraction in the cloud architecture. To make this approach scalable, we collect events independently within their layers, and aggregate on-demand.

Figure 1 shows Roots collecting data from all layers in a PaaS stack (i.e. full stack monitoring). Such monitoring enables it to trace the execution of individual requests through the cloud platform. In addition to Roots data collecting agents directly integrated with the cloud platform, Roots employs a collection of application benchmarking processes that periodically measure application latency. It uses these measurements to evaluate the performance SLOs for each application.

To avoid introducing delays to the application request processing flow, we implement all Roots data collecting agents as asynchronous tasks. That is, none of them suspend application request processing to store data. All expensive I/O tasks related to data collection and storage in Roots are executed out of the request processing flow. Roots collects all data into log files or memory buffers that are local to the components being monitored. This locally collected data is periodically sent to the data storage components of Roots using separate background tasks and batch communication operations. We also isolate PaaS services from potential failures in the Roots data collection or storage components to avoid cascading failures.

Roots data storage uses a database for persisting and querying monitoring data. We index key properties (application and timestamps) to optimize query performance. Roots also performs garbage collection for removal of old monitoring data when it is no longer useful to its anomaly detection algorithms.

3.2 Data Analysis

Roots data analysis components use two basic abstractions: *anomaly detectors* and *anomaly handlers*. Anomaly detectors are processes that periodically analyze the data collected for each application. Roots supports multiple detector implementations, where each implementation uses a different statistical method to look for performance anomalies. Detectors are configured per-application, making it possible for different applications to use different anomaly detectors. Roots also supports multiple concurrent anomaly detectors on the same application so that mixture approaches can be employed. Each anomaly detector has an execution schedule (e.g. run every 60 seconds), and a sliding window (e.g. from 10 minutes ago to now) associated with it. The boundaries of the window determines the scope of the data processed by the detector at any round of execution. Window is updated after each round of execution.

When an anomaly detector finds an anomaly in application performance, it sends an event to a collection of anomaly handlers. The event consists of a unique anomaly identifier, timestamp, application name, and the source detector's sliding window corresponding to the anomaly. Anomaly handlers are configured globally (i.e. each handler receives events from all detectors), but each handler can be configured to handle only certain types of events. Furthermore, handlers can fire their own events, which are also delivered to all listening anomaly handlers. Similar to detectors, Roots supports multiple anomaly handlers that support logging, alerting, dashboard updating, workload change detection, and bottleneck identification.

3.3 Roots Pods

Roots groups data analysis processes (i.e. anomaly detectors and handlers), and application benchmarking processes into units called *Roots pods*. Each Roots pod is responsible for starting and maintaining a collection of benchmarking and data analysis processes. Pods are self-contained entities, and there is no inter-communication between pods. Processes within a pod communicate via shared memory,

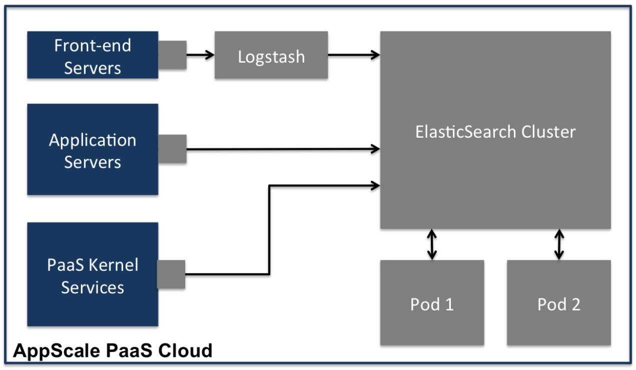

Figure 2: Roots prototype implementation for AppScale PaaS.

and call out to Roots data storage to retrieve performance measurements for analysis. Such encapsulation facilitates efficient starting/stopping of pods without impacting other PaaS and Roots components. Our pods design also facilitates replication for high availability and load balancing among multiple pods for scalability.

4. PROTOTYPE IMPLEMENTATION

To investigate the efficacy of Roots as an approach to implementing performance diagnostics as a PaaS service, we have developed a working prototype, and a set of algorithms that uses it to automatically identify SLO-violating performance anomalies. For this investigation, we integrate Roots into AppScale [22], an open source PaaS cloud that is API-compatible with the Google App Engine [13] public cloud. AppScale can run on bare metal or within cloud infrastructures (Amazon EC2, Eucalyptus, etc.), and executes GAE applications without modification. We minimally modify AppScale's internal components to integrate Roots.

Figure 2 shows an overview of our prototype implementation. Roots components are shown in grey, while the PaaS components are shown in blue. We use ElasticSearch [21] for data storage in our prototype. We configure AppScale's front-end server (Nginx) to tag all incoming application requests with a unique identifier via a custom HTTP header. All data collecting agents in the cloud extract this identifier, and include it as an attribute in all the events reported to ElasticSearch. This enables our prototype to aggregate events based on request IDs.

We implement a number of data collecting agents in AppScale to gather runtime information from all major components of the PaaS. These agents buffer data locally, and store them in ElasticSearch in batches. Roots persist events when the buffer accumulates 1MB of data or every 15 seconds, whichever comes first. This ensures that the events are promptly reported to the Roots data storage while keeping the memory footprint of the data collecting agents small and bounded. To capture the PaaS kernel invocation data, we augment AppScale's PaaS kernel implementation, which is derived from the GAE PaaS SDK. Specifically, we implement a Roots agent that monitors and times all PaaS kernel invocations, and reports them to ElasticSearch.

We implement Roots pods, which contain more computationally intensive Roots components, as standalone Java server processes. We use threads to run benchmarkers, anomaly detectors, and handlers concurrently within each pod. Pods communicate with ElasticSearch via a web API, and many of the data analysis tasks such as filtering and aggregation are performed in ElasticSearch.

4.1 Detecting SLO Violations

The SLO-based anomaly detector of Roots allows application developers to specify simple performance SLOs for deployed applications. A performance SLO is an upper bound on the application response time (T), and a probability (p) that the application response time is below the specified upper bound. When activated, the detector starts an application benchmarking process within a Roots pod that periodically measures the response time of the target application. Probes made by the benchmarking process are several seconds apart in time (defined by the process sampling rate). The detector periodically analyzes the collected response time measurements to check if the application meets the specified performance SLO. The detector also accepts a minimum sample count as a parameter. This is the minimum number of samples the detector should take before evaluating an SLO. If the fraction of response time measurements that are less than T falls below p, the SLO has been violated, and Roots triggers an anomaly event.

To prevent the detector from detecting the same anomaly multiple times, we flush the detection window upon each SLO violation. A side effect of this is that the detector is unable to detect another violation until the window fills again. For a sampling rate of 15 seconds and a minimum sample count of 100, this "warm up" period will be 25 minutes.

4.2 Workload Change Analyzer

To detect changes in workload, Roots implements a workload change analyzer as an anomaly handler. This handler is invoked for every anomaly detected by Roots. The workload change analyzer determines if an anomaly is due to a change in workload (i.e. an increase in the application request rate). Roots can report changes in workload via alerts; PaaS administrators can use such alerts to determine when to add resources to the system. Roots subjects anomalies *that are not due to a workload change* to bottleneck identification.

In contrast to the method described in [26, 25] which uses correlation between request latency and workload, our workload change analyzer uses change point detection algorithms to identify changes in the application's request rate. Our implementation of Roots supports a number of well known change point detection algorithms (PELT [20], binary segmentation and CL method [5]), any of which can be used to detect level shifts in the workload size. For the results in this paper, we use PELT to detect changes in workload.

4.3 Bottleneck Identification

When an anomaly occurs and it is not due to an increase in workload, the bottleneck identification feature attempts to find, across all the components executed by an application, the one that is most likely to have degraded application performance. PaaS applications consist of user code that is executed in one or more application servers, which makes remote service calls to PaaS kernel services. AppScale provides the same kernel services provided by GAE (datastore, memcache, urlfetch, blobstore, user management etc.). We consider each PaaS kernel invocation and the code running in the application server as separate *components*. Each application request causes one or more components to execute, and any one of the components can become a bottleneck to cause performance anomalies.

Roots tracks the total time and the time spent in individual components for each request, and relates them via the formula $T_{total} = T_{X_1} + T_{X_2} + ... + T_{X_n} + r$. T_{total} is the total execution time of a request. T_{X_i} is the time spent executing X_i; the i^{th} PaaS kernel invocation. r is the time spent in the resident application server executing user code (i.e. the time spent not executing PaaS services during a request). Roots measures the T values in the platform, and computes r using this formula. r is not measured directly because doing so would require application instrumentation. Given that typical PaaS-hosted web applications spend most of their time executing platform services [18], we expect $r \ll T_{X_1} + T_{X_2} + ... + T_{X_n}$ in the common case.

4.3.1 Selecting Bottleneck Candidates

Roots employs multiple candidate selection algorithms to identify components that are potentially the bottleneck responsible for the detected anomaly. Our algorithms look for a variety of inconsistencies and changes in the performance characteristics of individual components. In our prototype, we consider four selection algorithms: one that determines the relative importance (in terms of explained variance) of components, one that tracks variations in relative importance, and two distributional techniques that distinguish rare events and outliers in performance history.

Relative Importance. Relative importance [15] identifies the component that is contributing the most towards the variance in the total response time. To use it, Roots regresses the latencies of the PaaS kernel invocations that a specific request generates against the total response time of the request. That is, for a time window W just prior to the detection of the anomaly, Roots fits a linear model using linear regression of the form $T_{total} = T_{X_1} + T_{X_2} + ... + T_{X_n}$ over the per-request performance data in the window.

We omit r since it is typically small. For cases in which the anomaly occurs in r (e.g. a performance problem in the application server or user code), r may not fit our assumption that $r \ll T_{X_1} + T_{X_2} + ... + T_{X_n}$. When this occurs, we assume that r is normally distributed and independent, and filter out requests from the window for which r is larger than the 0.95 quantile ($r > \mu_r + 1.65\sigma_r$) to preclude them from skewing the regression. We consider the influence of r in other methods described below.

Roots ranks the regressors (i.e. T_{X_n} values) based on their contribution to the variance in T_{total}. It uses the LMG algorithm [24] to do so which is resistant to multicollinearity, and provides a breakdown of the R^2 value of the regression according to how strongly each regressor influences the variance of T_{total}. Roots selects the highest ranked component as a candidate.

Change in Relative Importance. The next method selects the most likely candidate by detecting *changes* in relative importance over time. This algorithm determines how the variance a component contributes towards the total response time has changed over time. For this method, Roots divides the time window W into equal-sized segments, and computes relative importance for regressors within each segment. This results in multiple time series of relative importance values. We subject each relative importance time series to change point analysis (PELT in our prototype), and identify the variable that shows a change and the greatest increase in relative importance. If such a variable is found, then the component associated with that variable is considered by Roots as a bottleneck candidate. The candidate selected by this method represents a component whose performance has been stable in the past, and has become variable recently.

High Quantiles. Roots next analyzes the empirical distributions of T_{X_k} and r. Out of all the available distributions, we wish to find the one whose quantile values are the largest. Specifically, we compute a high quantile (e.g. 0.99 quantile) for each distribution. The component, whose distribution contains the largest quantile value is chosen as another potential candidate for the bottleneck since it has recently had (i.e. within the analysis window) large latencies.

Tail End Values. Finally, Roots analyzes each T_{X_k} and r distribution to identify the one with the largest outlier value with respect to a particular high quantile. For each maximum latency value t, we compute the metric P_t^q as the percentage difference between t and a target quantile q of the corresponding distribution. We set q to 0.99 in our experiments. Roots selects the component with the distribution that has the largest P_t^q as another potential bottleneck candidate. This method identifies candidates that contain rare, high-valued outliers (point anomalies) in their distributions.

4.3.2 Selecting Among Candidates

The above four methods may select up to four candidate components for the bottleneck. We designate the candidate chosen by a majority of methods as the actual bottleneck. Ties are broken by assigning more priority to the candidate chosen by the relative importance method.

5. RESULTS

We next evaluate the efficacy of Roots as a performance monitoring and root cause analysis system for PaaS-hosted web applications. To do so, we consider its ability to identify and characterize SLO violations. For violations that are not caused by a change in workload, we evaluate Roots' ability to identify the PaaS component that is the cause of the performance anomaly. Finally, we investigate the performance and scalability of the Roots prototype.

5.1 SLO Anomaly Detection

We first experiment with the SLO-based anomaly detector of Roots using a simple HTML-producing, Java web application called "guestbook". This application allows users to login and post comments. It uses the AppScale datastore service to persist posted comments, and the AppScale user management service to handle authentication. Each request processed by guestbook results in two PaaS kernel service invocations – one to check if the user is logged in, and another to retrieve the existing comments from the datastore. We conduct all our experiments on a single node AppScale cloud except where specified. The node itself is an Ubuntu 14.04 VM with 4 virtual 2.4 GHz CPU cores and 4GB of memory, provisioned by a Eucalyptus IaaS cloud [31].

We run the SLO-based anomaly detector of Roots on guestbook with a sampling rate of 15 seconds, an analysis rate of 60 seconds, and a window size of 1 hour. We set

Table 1: Number of anomalies detected in guestbook app under different SLOs (L_1, L_2 and L_3) when injecting faults into two different PaaS kernel services.

Faulty Service	L_1 (30ms)	L_2 (35ms)	L_3 (45ms)
datastore	18	11	10
user management	19	15	10

the minimum samples count to 100, and run a series of experiments with different SLOs on the guestbook application. Specifically, we fix the SLO success probability at 95%, and set the response time upper bound to $\mu_g + n\sigma_g$. μ_g and σ_g represent the mean and standard deviation of the guestbook's response time. We learn these two parameters *a priori* by benchmarking the application. Then we obtain three different upper bound values for the guestbook's response time by setting n to 2, 3 and 5 and denote the resulting three SLOs L_1, L_2 and L_3 respectively.

Next, we inject performance faults into AppScale by modifying its code to cause the datastore service to be slow to respond. This fault injection logic activates once every hour, and slows down all datastore invocations by 45ms over a period of 3 minutes. We chose 45ms because it is equal to $\mu_g + 5\sigma_g$ for the AppScale deployment under test. Therefore this delay is sufficient to violate all three SLOs used in our experiments. We also perform experiments in which we inject faults into the user management service of AppScale. Each experiment is run for a period of 10 hours.

5.1.1 Anomaly Detection Accuracy

Table 1 shows how the number of anomalies detected by Roots in a 10 hour period varies when the SLO is changed. The number of anomalies drops noticeably when the response time upper bound is increased. When the L_3 SLO (45ms) is used, the only anomalies detected are the ones caused by our hourly fault injection mechanism. As the SLO is tightened by lowering the upper bound, Roots detects additional anomalies. These additional anomalies result from a combination of injected faults, and other naturally occurring faults in the system. That is, Roots detects naturally occurring faults (temporary spikes in application latency), while a number of injected faults are still in the sliding window of the anomaly detector. Together these two types of faults caused SLO violations usually several minutes after the fault injection period has expired. In each case, however, a manual inspection of the data reveals that Roots identified *all* of the injected and natural faults correctly, missing none subject to the tightness of the SLO.

5.1.2 Anomaly Detection Speed

Next we analyze how fast and often Roots can detect anomalies in an application. We consider the performance of guestbook under the L_1 SLO while injecting faults into the datastore service. Figure 3 shows anomalies detected by Roots as events on a time line. The horizontal axis represents passage of time. The red arrows indicate the start of a fault injection period, where each period lasts up to 3 minutes. The blue arrows indicate the Roots anomaly detection events. Note that every fault injection period is immediately followed by an anomaly detection event, implying near real time reaction from Roots. The only exception is the fault injection window at 20:00 hours, which is not immediately followed by an anomaly detection event. Roots detected another naturally occurring anomaly (i.e. one that we did not explicitly inject but nonetheless caused an SLO violation) at 19:52 hours, which caused the anomaly detector to go into the warm up mode. Therefore Roots did not immediately react to the faults injected at 20:00 hours. Roots detects the anomaly once it becomes active again at 20:17. As in the previous experiment, the detection accuracy is 100%, as determined by manual inspection.

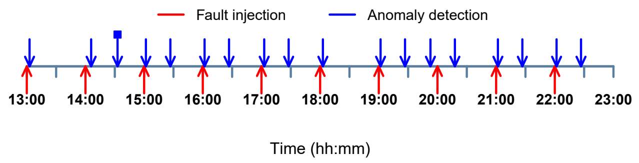

Figure 3: Anomaly detection in guestbook application during a period of 10 hours. Red arrows indicate fault injection at the datastore service. Blue arrows indicate all anomalies detected by Roots.

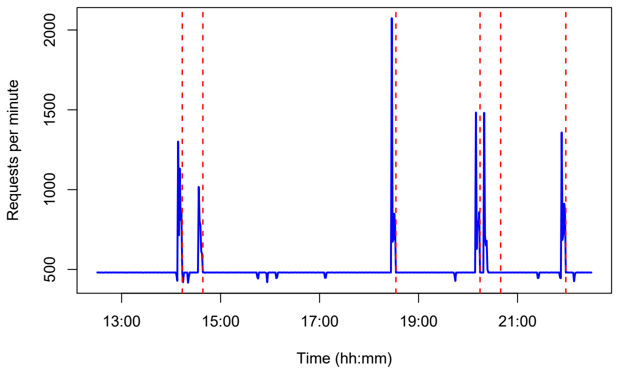

Figure 4: Workload size (requests per minute) over time for the key-value store application. The test client randomly sends large bursts of traffic causing the spikes in the plot. Roots anomaly detection events are shown in red dashed lines.

5.2 Workload Change Analyzer

Next we evaluate the Roots workload change analyzer. In this experiment we run a varying workload against a test application for 10 hours. The test application is an online key-value store that supports basic data management operations. The load generating client is programmed to maintain a mean workload level of 500 requests per minute. However, the client is also programmed to randomly send large bursts of traffic at times of its choosing. During these bursts the client may send more than 1000 requests a minute, thus impacting the performance of the application server that hosts the key-value store. Figure 4 shows how the application workload has changed over time. The workload generator has produced 6 large bursts of traffic during the period of the experiment, which appear as tall spikes in the plot. Note that each burst is immediately followed by a Roots anomaly detection event (shown by red dashed lines). In each of these 6 cases, the increase in workload caused a violation of

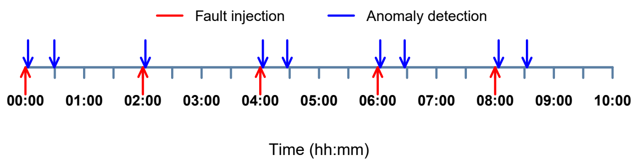

Figure 5: Anomaly detection in stock-trader application while injecting faults to the 1st kernel invocation

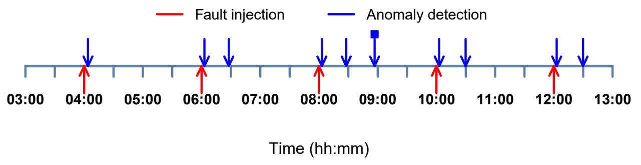

Figure 6: Anomaly detection in stock-trader application while injecting faults to the 2nd kernel invocation

the application performance SLO. Roots detected the corresponding anomalies, and determined them to be caused by changes in the workload size. As a result, bottleneck identification was not triggered for any of these anomalies. Even though the bursts of traffic appear to be momentary spikes, each burst lasts for 4 to 5 minutes thereby causing a lasting impact on the application performance. The PELT change point detection method used in this experimental set up is ideally suited for detecting such lasting changes in the workload level.

5.3 Bottleneck Identification

In this subsection, we evaluate the bottleneck identification capability of Roots. We first discuss the results obtained using the guestbook application, and follow with results obtained using a more complex application. In the experimental run illustrated in figure 3, Roots determined that all the detected anomalies except for one were caused by the AppScale datastore service. This is consistent with our expectations since in this experiment we artificially inject faults into the datastore. The only anomaly that is not traced back to the datastore service is the one that was detected at 14:32 hours. This is indicated by the blue arrow with a square marker at the top. For this anomaly, Roots concluded that the bottleneck is the local execution at the application server (r). We have verified this result by manually inspecting the AppScale logs and traces of data collected by Roots. As it turns out, between 14:19 and 14:22 the application server hosting the guestbook application experienced some problems, which caused request latency to increase significantly. Therefore we can conclude that Roots has correctly identified the root causes of all 18 anomalies in this experimental run including one that we did not inject explicitly.

5.4 A More Complex Example

To evaluate how the bottleneck identification performs when an application makes more than 2 PaaS kernel invocations, we conduct another experiment using an application called “stock-trader”. This application allows setting up organizations, and simulating trading of stocks between the organizations. The two main operations in this application are

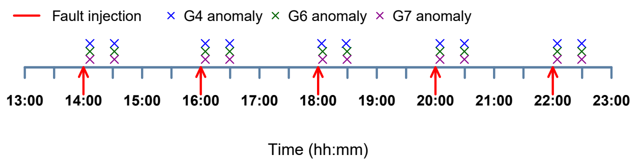

Figure 7: SLO-violation Anomaly detection in 8 applications deployed in a clustered AppScale cloud.

buy and *sell*. Each of these operations makes 8 calls to the AppScale datastore. According to our previous work [18], 8 kernel invocations in the same path of execution is very rare in web applications developed for a PaaS cloud. The probability of finding an execution path with more than 5 kernel invocations in a sample of PaaS-hosted applications is less than 1%. Therefore the stock-trader application is a good extreme case example to test the Roots bottleneck identification support. In this experiment we configure the anomaly detector to check for the response time SLO of 177ms with 95% success probability.

In one of our experimental runs we inject faults into the first datastore query executed by the buy operation of stock-trader. The fault injection logic runs every two hours, and lasts for 3 minutes. The duration of the full experiment is 10 hours. Figure 5 shows the resulting event sequence. Note that every fault injection event is immediately followed by a Roots anomaly detection event. There are also four additional anomalies in the time line which were SLO violations caused by a combination of injected faults, and naturally occurring faults in the system. For all the anomalies detected in this test, Roots correctly selected the first datastore call in the application code as the bottleneck. The additional four anomalies occurred because a large number of injected faults were in the sliding window of the detector. Therefore, it is accurate to attribute those anomalies also to the first datastore query of the application.

Figure 6 shows the results from a similar experiment where we inject faults into the second datastore query executed by the operation. Here also Roots detects all the artificially induced anomalies along with a few extras. All the anomalies, except for one, are determined to be caused by the second datastore query of the buy operation. The anomaly detected at 08:56 (marked with a square on top of the blue arrow) is attributed to the fourth datastore query executed by the application. We have manually verified this conclusion to be accurate. Since 08:27 (when the previous anomaly was detected), the fourth datastore query has frequently taken a long time to execute (again, on its own), which resulted in an SLO violation at 08:56 hours.

5.5 Multi-tenant Cluster Setting

To demonstrate how Roots can be used in a multi-node environment, we deploy AppScale using a cluster of 10 virtual machines (VMs). VMs are provisioned by a Eucalyptus IaaS cloud, and each VM is comprised of 2 CPU cores and 2GB memory. Then we proceed to deploy 8 instances of the guestbook application on AppScale. We use the multi-tenant support in AppScale to register each instance of guestbook as a different application ($G1$ through $G8$). Each application instance has its own datastore namespace, is isolated from the others, and is accessed via its own public URL. We disable auto-scaling support in the AppScale cloud, and inject faults

Table 2: Summary of Roots efficacy results.

Feature	Results Observed in Roots
Detecting accuracy	The first occurrence of all artificially induced anomalies were detected. Roots also detected several "natural" anomalies in AppScale.
Anomaly Differentiation	All anomalies were correctly identified as either due to workload change or bottleneck.
Bottleneck Identification	In all the cases where a bottleneck was identified Roots identified the correct bottleneck.
Reaction time	All induced SLO violations were detected as soon as enough samples of the fault were taken by the benchmarking process.

into the datastore service of AppScale by delaying queries from a particular VM by 100ms. We identify the VM by its IP address in our test environment, and shall refer to it as V_f in the discussion. We trigger the fault injection every 2 hours for a duration of 5 minutes. We then monitor the applications using Roots for a period of 10 hours. Each anomaly detector is configured to check for the 75ms response time SLO with 95% success rate. ElasticSearch, Logstash and the Roots pod are deployed on a separate VM.

Figure 7 shows the resulting event sequence. Note that we detect anomalies in 3 applications ($G4$, $G6$ and $G7$) immediately after each fault injection. Inspecting the topology of our AppScale cluster revealed that these were the only 3 applications that were hosted on V_f. As a result, bi-hourly fault injection caused their SLOs to get violated. Other applications did not exhibit any SLO violations since the SLO specifies a high response time upper bound. In each case Roots detected the SLO violations 2-3 minutes into the fault injection period. As soon as this happens, the anomaly detectors of $G4$, $G6$ and $G7$ entered their warmup phases. Our fault injection logic continued to inject faults for at least 2 more minutes. Therefore when the anomaly detectors reactivated after 25 minutes (which is the time to collect the minimum sample count), they each saw another SLO violation. Hence another set of detection events appears in the figure approximately half an hour after the original fault injection events.

5.6 Results Summary

Table 2 summarizes our results. While not exhaustive, these results indicate that Roots is able to achieve a high degree of accuracy in detecting and characterizing performance anomalies. Moreover, for the experimental configurations that we consider, Roots is able to accurately identify the component that causes each anomaly in a production-quality, open source PaaS.

5.7 Roots Performance and Scalability

We evaluate the performance overhead incurred by Roots on the applications deployed in the cloud platform. We are particularly interested in understanding the overhead of recording the PaaS kernel invocations made by each application, since this feature requires some changes to the PaaS kernel implementation. We deploy a number of applications on a vanilla AppScale cloud (with no Roots), and measure their request latencies. We use the popular Apache Bench tool to measure the request latency under a varying number of concurrent clients. We then take the same measurements on an AppScale cloud with Roots, and compare the results against the ones obtained from the vanilla AppScale cloud. In both environments we disable the auto-scaling support of AppScale, so that all client requests are served from a single application server instance. In our prototype implementation of Roots, the kernel invocation events get buffered in the application server before they are sent to the Roots data storage. We wish to explore how this feature performs when the application server is under heavy load.

Table 3: Latency comparison of applications when running on a vanilla AppScale cloud vs when running on a Roots-enabled AppScale cloud.

	Without Roots		With Roots	
App./Concurrency	Mean (ms)	SD	Mean (ms)	SD
guestbook/50	375	51.4	374	53.0
stock-trader/50	3631	690.8	3552	667.7
kv store/50	169	26.7	150	25.4

Table 3 shows the comparison of request latencies with 50 clients for each of the applications. The data shows that Roots does not add a significant overhead to the request latency in any of the scenarios considered. In all the cases, the mean request latency when Roots is in use is within one standard deviation from the mean request latency when Roots is not in use.

Finally, to demonstrate Roots scalability, we deploy a Roots pod on a virtual machine with 4 CPU cores and 4GB memory. With 10000 concurrent detectors, the maximum memory usage of the pod was 778MB and the CPU usage was 60% of a single core. Using just the one machine, with 4 cores and 4GB of memory, a single pod was able to service 40000 concurrent detectors. Since one pod can service a large number of applications simultaneously and pods are independent, these results indicate that Roots scales in the number of detectors.

6. RELATED WORK

Roots falls into the category of performance anomaly detection and bottleneck identification (PADBI) systems [17]. They play a crucial role in achieving guaranteed performance and quality of service by detecting performance issues before they escalate into major outages or SLO violations [16]. However, the paradigm of cloud computing is yet to be fully penetrated by PADBI systems research. The size, complexity and the dynamic nature of cloud platforms make performance monitoring a particularly challenging problem. The existing technologies like Amazon CloudWatch [7], New Relic [29] and DataDog [9] facilitate monitoring cloud applications by instrumenting low level cloud resources (e.g. virtual machines), and application code. But such technologies are either impracticable or insufficient in PaaS clouds where the low level cloud resources are hidden from users by software abstractions, while application-level instrumentation is generally tedious and error-prone.

Tracing system activities via request tagging has been employed in several previous works such as X-Trace [12] and PinPoint [6]. X-Trace records network activities across protocols and layers, but does not support root cause analysis. PinPoint traces interactions among J2EE middleware components to localize faults rather than performance issues. Currently it is limited to single-machine tracing within a JVM.

Aguilera et al developed a performance debugging framework for distributed systems comprised of blackbox components [1]. They infer the inter-call causal paths between application components, and attribute delays to those components. Roots does something similar, but we only consider one level of interaction in our work – from application code to PaaS kernel services. This is sufficient to determine the PaaS kernel invocation that may have caused a performance anomaly. The blackbox components considered in our work (i.e. the PaaS kernel services) do not typically interact with each other. However, each PaaS kernel service is a distributed system of its own right with many internal components running on different containers and VMs. In our future work we wish to also factor in the events that occur within PaaS services and the underlying IaaS components, and expand the bottleneck identification capabilities of Roots down to internal service components, containers and VMs.

Attariyan et al presented X-ray [3], a performance summarization system that can be applied to a wide range of production software. X-ray attributes performance costs to each basic block executed by a software application, and estimates the likelihood a block was executed due to a potential root cause. With that X-ray can compute a list of root cause events ordered by performance costs. However, X-ray relies on instrumenting application binaries. It also attributes performance issues to root causes under user's control such as configuration issues. Therefore for each application the X-ray user must specify the application configurations, program inputs, as well as a way to identify when a new request begins. Roots requires neither binary instrumentation, nor any additional user input. Also it can detect performance bottlenecks in the cloud platform that are beyond application developer's control. In the future we wish to explore the idea of selective instrumentation, in which we temporarily instrument an application to gather additional runtime data for a period, so we can perform X-ray like performance summarization on-demand.

Systems noted above ([6, 1, 3]) collect/trace data online, but perform heavy computations offline. We take the same approach in Roots by running anomaly detection and root cause analysis as offline processes. However, none of the above systems have been designed for or tested in a cloud environment. Roots on the other hand is specifically designed to monitor web applications in PaaS clouds where the abstractions, scale and user expectations pose unique challenges.

Our work is heavily inspired by the past literature that detail the key features of cloud APMs [8, 17] . We also borrow from Magalhaes and Silva who uses statistical correlation and linear regression to perform root cause analysis in web applications [26, 25]. Dean et al implemented PerfCompass [10], an anomaly detection and localization method for IaaS clouds. They instrument operating system kernels of VMs to perform root cause analysis in IaaS clouds. We take a similar approach where we instrument the kernel services of the PaaS cloud. Nguyen et al presented PAL, another anomaly detection mechanism targeting distributed applications deployed on IaaS clouds [30]. Similar to Roots, they also use an SLO monitoring approach to detect anomalies.

Anomaly detection is a general problem not restricted to performance analysis. Researchers have studied anomaly detection from various standpoints and come up with many algorithms [4]. While we use many statistical methods in our work (change point analysis, relative importance, quantile analysis), Roots is not tied to any of these techniques. Rather, we provide an extensible framework on top of which new anomaly detectors and anomaly handlers can be built.

7. CONCLUSIONS AND FUTURE WORK

We propose Roots, a near real time monitoring and diagnostics framework for web applications deployed in a PaaS cloud. Roots is designed to function as a curated service built into the cloud platform, as opposed to an external monitoring system. It relieves the application developers from having to configure their own monitoring solutions or, indeed, from having to instrument the application code.

We evaluate Roots using a prototype built for the AppScale open source PaaS. Our results indicate that Roots is effective at detecting performance anomalies in near real time. We also show that our bottleneck identification algorithm produces accurate results nearly 100% of the time (cf Table 2), pinpointing the exact PaaS service or the application component responsible for each anomaly. Our empirical trials further reveal that Roots does not add a significant overhead to the applications deployed on the cloud platform. Finally, we show that Roots is lightweight, and scales well to handle large populations of applications.

As part of future work, we plan to expand the data gathering capabilities of Roots into the low level virtual machines and containers that host various services of the cloud platform. We intend to tap into the hypervisors and container managers to harvest runtime data regarding the resource usage (CPU, memory, disk etc.) of PaaS services and other application components. Roots will use this information to identify performance anomalies that are caused by the cloud resources on which the PaaS relies.

8. ACKNOWLEDGMENTS

This work is funded in part by NSF (CCF-1539586, ACI-1541215, CNS-1218808, CNS-0905237, ACI-0751315), NIH (1R01EB014877-01), ONR NEEC (N00174-16-C-0020), Huawei Technologies, and the California Energy Commission (PON-14-304).

9. REFERENCES

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[2] N. Antonopoulos and L. Gillam. *Cloud Computing: Principles, Systems and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[3] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of*

the 10th USENIX Conference on Operating Systems Design and Implementation, 2012.

[4] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), 2009.

[5] C. Chen and L.-M. Liu. Joint estimation of model parameters and outlier effects in time series. *Journal of the American Statistical Association*, 88(421):284–297, 1993.

[6] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002.

[7] Amazon cloud watch, 2016. `https://aws.amazon.com/cloudwatch` [Accessed Sep 2016].

[8] G. Da Cunha Rodrigues, R. N. Calheiros, V. T. Guimaraes, G. L. d. Santos, M. B. de Carvalho, L. Z. Granville, L. M. R. Tarouco, and R. Buyya. Monitoring of cloud computing environments: Concepts, solutions, trends, and future directions. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016.

[9] Datadog: Cloud monitoring as a service, 2016. `https://www.datadoghq.com` [Accessed Sep 2016].

[10] D. J. Dean, H. Nguyen, P. Wang, and X. Gu. Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds. In *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*, 2014.

[11] Dynatrace: Digital performance management and application performance monitoring, 2016. `https://www.dynatrace.com` [Accessed Sep 2016].

[12] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, 2007.

[13] App Engine - Run your applications on a fully managed PaaS, 2015. `https://cloud.google.com/appengine` [Accessed March 2015].

[14] Google Cloud SDK Service Quotas, 2015. `https://cloud.google.com/appengine/docs/quotas` [Accessed March 2015].

[15] U. Groemping. Relative importance for linear regression in r: The package relaimpo. *Journal of Statistical Software*, 17(1), 2006.

[16] Q. Guan, Z. Zhang, and S. Fu. Proactive failure management by integrated unsupervised and semi-supervised learning for dependable cloud systems. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, 2011.

[17] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth. Performance anomaly detection and bottleneck identification. *ACM Comput. Surv.*, 48(1), July 2015.

[18] H. Jayathilaka, C. Krintz, and R. Wolski. Response time service level agreements for cloud-hosted web applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[19] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *J. Netw. Syst. Manage.*, 11(1), Mar. 2003.

[20] R. Killick, P. Fearnhead, and I. A. Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500):1590–1598, 2012.

[21] O. Kononenko, O. Baysal, R. Holmes, and M. W. Godfrey. Mining modern repositories with elasticsearch. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[22] C. Krintz. The appscale cloud platform: Enabling portable, scalable web application deployment. *IEEE Internet Computing*, 17(2), 2013.

[23] Latency is Everywhere and it Costs Your Sales, 2009. `http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it` [Accessed Sep 2016].

[24] G. R. Lindeman R.H., Merenda P.F. *Introduction to Bivariate and Multivariate Analysis*. Scott, Foresman, Glenview, IL, 1980.

[25] J. a. P. Magalhães and L. M. Silva. Root-cause analysis of performance anomalies in web-based applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011.

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

[27] Microsoft Azure Cloud SDK Service Quotas, 2015. `http://azure.microsoft.com/en-us/documentation/articles/azure-subscription-service-limits` [Accessed March 2015].

[28] M. Natu, R. K. Ghosh, R. K. Shyamsundar, and R. Ranjan. Holistic performance monitoring of hybrid clouds: Complexities and future directions. *IEEE Cloud Computing*, 3(1), Jan 2016.

[29] New relic: Application performance management and monitoring, 2016. `https://newrelic.com` [Accessed Sep 2016].

[30] H. Nguyen, Y. Tan, and X. Gu. Pal: Propagation-aware anomaly localization for cloud hosted distributed applications. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, 2011.

[31] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009.

[32] P. Pinheiro, M. Aparicio, and C. Costa. Adoption of cloud computing systems. In *Proceedings of the International Conference on Information Systems and Design of Communication*, 2014.

[33] M. Soni. Cloud computing basics—platform as a service (paas). *Linux J.*, 2014(238), 2014.