University of California
Santa Barbara

Governance of Cloud-hosted Web Applications

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Hiranya K. Jayathilaka
(Iyagalle Gedara)

Committee in charge:

Professor Chandra Krintz, Chair
Professor Rich Wolski
Professor Tevfik Bultan

December 2016

The Dissertation of Hiranya K. Jayathilaka is approved.

Professor Rich Wolski

Professor Tevfik Bultan

Professor Chandra Krintz, Committee Chair

December 2016

Governance of Cloud-hosted Web Applications

by

Hiranya K. Jayathilaka

Acknowledgements

My sincere gratitude goes out to my advisors Chandra Krintz, Rich Wolski and Tevfik Bultan, whose guidance has been immensely helpful to me during my time at graduate school. They are some of the most well informed and innovative people I know, and I consider it both an honor and a privilege to have had the opportunity to work under their tutelage.

I also thank my wonderful parents, whose love and continued support have been the foundation of my life. Thank you for the values you have instilled in me, and giving me courage to face all sorts of intellectual and emotional challenges.

I am also grateful to all my teachers and mentors from the past, who generously shared their knowledge and experiences with me. My special thanks go out to Sanjiva Weerawarana whose intelligence and leadership skills inspire me to be better at my work everyday.

Finally, I thank the amazing faculty, staff and the community at UC Santa Barbara, whose smiles and positive attitude have made my life so much easier and exciting. I especially like to mention my colleagues at the RACELab, both present and past, for all the intellectual stimulation as well as their warm sense of friendship.

Thank you, and Ayubowan!

Curriculum Vitæ

Hiranya K. Jayathilaka

Education

2016	Ph.D. in Computer Science (Expected), University of California, Santa Barbara, United States.
2009	B.Sc. Engineering (Hons) Degree, University of Moratuwa, Sri Lanka.

Publications

Service-Level Agreement Durability for Web Service Response Time
H. Jayathilaka, C. Krintz, R. Wolski
International Conference on Cloud Computing Technology and Science (CloudCom), 2015.

Response time service level agreements for cloud-hosted web applications
H. Jayathilaka, C. Krintz, and R. Wolski
ACM Symposium on Cloud Computing (SoCC), 2015.

EAGER: Deployment-Time API Governance for Modern PaaS Clouds
H. Jayathilaka, C. Krintz, and R. Wolski
IC2E Workshop on the Future of PaaS, 2015.

Using Syntactic and Semantic Similarity of Web APIs to Estimate Porting Effort
H. Jayathilaka, A. Pucher, C. Krintz, and R. Wolski
International Journal of Services Computing (IJSC), 2014.

Towards Automatically Estimating Porting Effort between Web Service APIs
H. Jayathilaka, C. Krintz, and R. Wolski
International Conference on Services Computing (SCC), 2014.

Cloud Platform Support for API Governance
C. Krintz, H. Jayathilaka, S. Dimopoulos, A. Pucher, R. Wolski, and T. Bultan
IC2E Workshop on the Future of PaaS, 2014.

Service-driven Computing with APIs: Concepts, Frameworks and Emerging Trends
H. Jayathilaka, C. Krintz, and R. Wolski
IGI Global Handbook of Research on Architectural Trends in Service-driven Computing,

2014.

Improved Server Architecture for Highly Efficient Message Mediation
H. Jayathilaka, P. Fernando, P. Fremantle, K. Indrasiri, D. Abeyruwan, S. Kamburugamuwa, S. Jayasumana, S. Weerawarana and S. Perera
International Conference on Information Integration and Web-based Applications and Services (IIWAS), 2013.

Extending Modern PaaS Clouds with BSP to Execute Legacy MPI Applications
H. Jayathilaka and M. Agun
ACM Symposium on Cloud Computing (SoCC), 2013.

Abstract

Governance of Cloud-hosted Web Applications

by

Hiranya K. Jayathilaka

Cloud computing has revolutionized the way developers implement and deploy applications. By running applications on large-scale compute infrastructures and programming platforms that are remotely accessible as utility services, cloud computing provides scalability, high-availability, and increased user productivity.

Despite the advantages inherent to the cloud computing model, it has also given rise to several software management and maintenance issues. Specifically, cloud platforms do not enforce developer best practices, and other administrative requirements when deploying applications. Cloud platforms also do not facilitate establishing service level objectives (SLOs) on application performance, which are necessary to ensure reliable and consistent operation of applications. Moreover, cloud platforms do not provide adequate support to monitor the performance of deployed applications, and conduct root cause analysis when an application exhibits a performance anomaly.

We employ *governance* as a methodology to address the above mentioned issues prevalent in cloud platforms. We devise novel governance solutions that achieve administrative conformance, developer best practices, and performance SLOs in the cloud via policy enforcement, SLO prediction, performance anomaly detection and root cause analysis. The proposed solutions are fully automated, and built into the cloud platforms as cloud-native features thereby precluding the application developers from having to implement similar features by themselves. We evaluate our methodology using real world cloud platforms, and show that our solutions are highly effective and efficient.

Contents

Chapter 1

Introduction

Cloud computing turns compute infrastructures, programming platforms and software systems into online utility services that can be easily shared among many users [1, 2]. It enables processing and storing data on large, managed infrastructures and programming platforms, that can be accessed remotely via the internet. This provides an alternative to running applications on local servers, personal computers, and mobile devices, all of which have strict resource constraints. Today, cloud computing technologies can be obtained from a large and growing number of providers. Some of these providers offer hosted cloud platforms that can be used via the web to deploy applications without installing any physical hardware (e.g. Amazon AWS [3], Google App Engine [4], Microsoft Azure [5]). Others provide cloud technologies as downloadable software, which users can install on their computers or data centers to set up their own private clouds (e.g. Eucalyptus [6], AppScale [7], OpenShift [8]).

Cloud computing model provides high scalability, high availability and enhanced levels of user productivity. Cloud platforms run on large resource pools, typically in one or more data centers managed by the platform provider. Therefore cloud platforms have access to a vast amount of hardware and software resources. This enables cloud-hosted applications to scale to varying load conditions, and maintain high availability. Moreover, by offering resources as utility services, cloud computing is able to facilitate a cost-effective, on-demand resource provisioning model that greatly enhances user productivity.

Over the last decade cloud computing technologies have enjoyed explosive growth, and near universal adoption due to their many benefits and promises [9, 10]. Industry analysts project that the cloud computing market value will exceed $150 billion by the year 2020 [11]. A large number of organizations run their entire business as a cloud-based operation (e.g. Netflix, Snapchat). For startups and academic researchers who do not have a large IT budget or a staff, the cost-effective on-demand resource provisioning model of the cloud has proved to be indispensable. The growing number of academic conferences and journals dedicated to discussing cloud computing is further evidence that cloud is an essential branch in the field of computer science.

Despite its many benefits, cloud computing has also given rise to several application development and maintenance challenges that have gone unaddressed for many years. As the number of applications deployed in cloud platforms continue to increase these shortcoming are rapidly becoming conspicuous. We highlight three such issues.

Firstly, cloud platforms lack the ability to enforce developer best practices and administrative conformance on deployed user applications. The developer best practices are the result of decades of software engineering research, and include code reuse, proper versioning of software artifacts, dependency management between application components, and backward compatible software updates. Administrative conformance refers to complying with various development and maintenance standards that an organization may wish to impose on all of their production software. Cloud platforms do not provide any facilities that enforce such developer practices or administrative standards. Instead, cloud platforms make it extremely trivial and quick to deploy new applications or update existing applications (i.e. roll out new versions). The resulting speed-up of the development cycles combined with the lack of oversight and verification, makes it extremely difficult for IT personnel to manage large volumes of cloud-hosted applications.

Secondly, today's cloud platforms do not provide support for establishing service level objectives (SLOs) regarding the performance of deployed applications. A performance SLO specifies a bound on application's response time (latency). Such bounds are vital for developers who implement downstream systems that consume the cloud-hosted applications, and cloud administrators who wish to maintain a consistent quality of service level. However, when an application is implemented for a cloud platform, one must subject it to extensive performance testing in order to comprehend its performance bounds; a process that is both tedious and time consuming. The difficulty in understanding the performance bounds of cloud-hosted applications is primarily due to the very high level of abstraction provided by the cloud platforms. These abstractions shield many details concerning the application runtime, and without visibility into such low level application execution details it is impossible to build a robust performance model for a cloud-hosted application. Due to this reason, it is not possible to stipulate SLOs on the performance of cloud-hosted applications. Consequently, existing cloud platforms only offer SLOs regarding service availability.

Thirdly, cloud platforms do not provide adequate support for monitoring application performance, and running diagnostics when an application fails to meet its performance SLOs. Most cloud platforms only provide the simplest monitoring and logging features, and do not provide any mechanisms for detecting performance anomalies or identifying bottlenecks in the application code or the underlying cloud platform. This limitation has given rise to a new class of third party service providers that specialize in monitoring cloud applications (e.g. New Relic [12], Dynatrace [13], Datadog [14]). But these third party solutions are expensive. They also require code instrumentation, which if not done correctly, leads to incorrect diagnoses. The perturbation introduced by the instrumentation also changes and degrades application performance. Furthermore, the extrinsic monitoring systems have a restricted view of the cloud platform, due to the high level of abstraction provided by cloud platform software. Therefore they cannot observe the complexity of the cloud platform in full, and hence cannot pinpoint the component that might be responsible for a perceived application performance anomaly.

In order to make the cloud computing model more dependable, maintainable and convenient for the users as well as the cloud service providers, the above limitations need to be addressed satisfactorily. Doing so will greatly simplify the tasks of developing cloud applications, and maintaining them in the long run. Developers will be able to specify SLOs on the performance of their cloud-hosted applications, and offer competitive service level agreements (SLAs) to the end users that consume those applications. Developers as well as cloud administrators will be able to detect performance anomalies promptly, and take corrective actions before the issues escalate to major outages or other crises.

Our research focuses on addressing the above issues in cloud environments using *governance*. We define governance as the mechanism by which the acceptable operational parameters are specified and maintained in a software system [15, 16]. This involves multiple steps:

- Specifying the acceptable operational parameters
- Enforcing the specified parameters
- Monitoring the system to detect deviations from the acceptable behavior

To learn the feasibility and the efficacy of applying governance techniques in a cloud platform, we propose and explore the following thesis question: Can we efficiently enforce governance for cloud-hosted web applications to achieve administrative conformance, developer best practices, and performance SLOs through automated analysis and diagnostics?

For governance to be useful within the context of cloud computing, it must be both efficient and automated. Cloud platforms are comprised of many components that have different life cycles and maintenance requirements. They also serve a very large number of users who deploy applications in the cloud. Therefore governance systems designed for the cloud should scale to handle a large number of applications and related software components, without introducing a significant runtime overhead on them. Also they must be fully automated since it is not practical for a human administrator to be involved in the governance process given the scale of the cloud platforms.

Automated governance for software systems is a well researched area, especially in connection with classic web services and service-oriented architecture (SOA) applications [16, 17, 18, 19, 20]. We adapt the methodologies outlined in the existing SOA governance research corpus, so they can be applied to cloud computing systems. These methodologies enable specifying acceptable behavior via machine readable policies, which are then automatically enforced by a policy enforcement agent. Monitoring agents watch the system to detect any deviations from the acceptable behavior (i.e. policy violations), and alert users or follow predefined corrective procedures. We can envision similar facilities being implemented in a cloud platform to achieve administrative conformance, developer best practices and performance SLOs. The operational parameters in this case may include coding and deployment conventions for the cloud-hosted applications, and their expected performance levels.

In order to answer the above thesis question by developing efficient, automated governance systems, we take the following three-step approach.

- Design and implement a scalable, low-overhead governance framework for cloud platforms, complete with a policy specification language and a policy enforcer. The governance framework should be built into the cloud platforms, and must keep the runtime overhead of the user applications to a minimum while enforcing developer best practices and administrative conformance.

- Design and implement a methodology for formulating performance SLOs (bounds) for cloud-hosted web applications, without subjecting them to extensive performance testing or instrumentation. The formulated SLOs must be correct, tight and durable in the face of changing conditions of the cloud.
- Design and implement a scalable cloud application performance monitoring (APM) framework for detecting violations of performance SLOs. For each violation detected, the framework should be able to run diagnostics, and identify the potential root cause. It should support collecting data from the cloud platform without instrumenting user code, and without introducing significant runtime overheads.

To achieve administrative conformance and developer best practices with minimal overhead, we perform governance policy enforcement when an application is deployed; a technique that we term deployment-time policy enforcement. We explore the trade off between what policies can be enforced, and when they can be enforced with respect to the life cycle of a cloud-hosted application. We show that not all policies are enforceable at deployment-time, and therefore some support for run-time policy enforcement is also required in the cloud. However, we find that deployment-time policy enforcement is efficient, and a governance framework that performs most, if not all, enforcement tasks at deployment-time can scale to thousands of applications and policies.

We combine static analysis with platform monitoring to establish performance SLOs for cloud-hosted applications. Static analysis extracts the sequence of critical operations (cloud services) invoked by a given application. Platform monitoring facilitates constructing a historic performance model for the individual operations. We then employ a time series analysis method to combine these results, and calculate statistical bounds for application response time. The performance bounds calculated in this manner are associated with a specific correctness probability, and hence can be used as SLOs. We also devise a statistical framework to evaluate the validity period of calculated performance bounds.

In order to detect and diagnose performance SLO violations, we monitor various performance events that occur in the cloud platform, correlate them, and employ statistical analysis to identify anomalous patterns. Any given statistical method is only sensitive to a certain class of anomalies. Therefore, to be able to diagnose a wide range of performance anomalies, we devise an algorithm that combines linear regression, change point detection and quantile analysis. Our approach detects performance SLO violations in near real time, and identifies the root cause of each event as a workload change or a performance bottleneck in the cloud platform. In case of performance bottlenecks, our approach also correctly identifies the exact component in the cloud platform, in which the bottleneck manifested.

Our contributions push the state of the art in cloud computing significantly towards achieving administrative conformance, developer best practices and performance SLOs. Moreover, our work addresses all the major steps associated with software system governance – specification, enforcement and monitoring. We show that this approach can significantly improve cloud platforms in terms of their reliability, developer-friendliness and ease of management. We also demonstrate that the governance capabilities proposed in our work can be built into existing cloud platforms, without having to implement them from the scratch.

Chapter 2

Background

2.1 Cloud Computing

Cloud computing is a form of distributed computing that turns compute infrastructure, programming platforms and software systems into scalable utility services [1, 2]. By exposing various compute and programming resources as utility services, cloud computing promotes resource sharing at scale via the Internet. The cloud model precludes the users from having to set up their own hardware, and in some cases also software. Instead, the users can simply acquire the resources "in the cloud" via the internet, and relinquish them when the resources are no longer needed. The cloud model also does not require the users to spend any start up capital. The users only have to pay for the resources they acquired, usually based on a pay-per-use billing model. Due to these benefits associated with cloud computing, many developers and organizations use the cloud as their preferred means of developing and deploying software applications [9, 10, 11].

Depending on the type of resources offered as services, cloud computing platforms can be categorized into three main categories [2].

Infrastructure-as-a-Service clouds (IaaS) Offers low-level compute, storage and networking resources as a service. Compute resources are typically provided in the form of on-demand virtual machines (VMs) with specific CPU, memory and disk configurations (e.g. Amazon EC2 [21], Google Compute Engine [22], Eucalyptus [23]). The provisioned VMs usually come with a base operating system installed. The users must install all the application software necessary to use them.

Platform-as-a-Service clouds (PaaS) Offers a programming platform as a service, that can be used to develop and deploy applications at scale (e.g. Google App Engine [4], AppScale [7], Heroku [24], Amazon Elastic Beanstalk [25]). The programming platform consists of several scalable services that can be used to obtain certain application features such as data storage, caching and authentication.

Software-as-a-Service clouds (SaaS) Offers a collection of software applications and tools as a service, that can be directly consumed by application endusers (e.g. Salesforce [26], Workday [27], Citrix go2meeting [28]). This can be thought of as a new way of delivering software to endusers. Instead of prompting the users to download and install any software, SaaS enables the users to consume software via the Internet.

Cloud-hosted applications expose one or more web application programming interfaces (web APIs) through which client programs can remotely interact with the applications. That is, clients send HTTP/S requests to the API, and receive machine readable responses (e.g. HTML, JSON, XML, Protocol Buffers [29]) in return. This type of web-accessible, cloud-hosted applications tend to be highly interactive, and clients have strict expectations on the application response time [30].

A cloud-hosted application may also consume web APIs exposed by other cloud-hosted applications. Thus, cloud-hosted applications form an intricate graph of inter-dependencies among them, where each application can service a set of client applications, while being dependent on a set of other applications. However, in general, each cloud-hosted application directly depends on the core services offered by the underlying cloud platform for compute power, storage, network connectivity and scalability.

In the next section we take a closer look at a specific type of cloud platforms – Platform-as-a-Service clouds. We use PaaS clouds as a case study and a testbed in a number of our explorations.

2.2 Platform-as-a-Service Clouds

PaaS clouds, which have been growing in popularity [31, 32], typically host web-accessible (HTTP/S) applications, to which they provide high levels of scalability, availability, and sandboxed execution. PaaS clouds provide scalability by automatically allocating resources for applications on the fly (auto scaling), and provide availability through the execution of multiple instances of the application. Applications deployed on a PaaS cloud depend on a number of scalable services intrinsic to the cloud platform. We refer to these services as *kernel services.*

PaaS clouds, through their kernel services, provide a high level of abstraction to the application developer that effectively hides all the infrastructure-level details such as physical resource allocation (CPU, memory, disk etc), operating system, and network configuration. Moreover, PaaS clouds do not require the developers to set up any utility services their applications might require such as a database or a distributed cache. Everything an application requires is provisioned and managed by the PaaS cloud. This enables application developers to focus solely on the programming aspects of their applications, without having to be concerned about deployment issues. On the other hand, the software abstractions provided by PaaS clouds obscure runtime details of applications making it difficult to reason about application performance, and diagnose performance

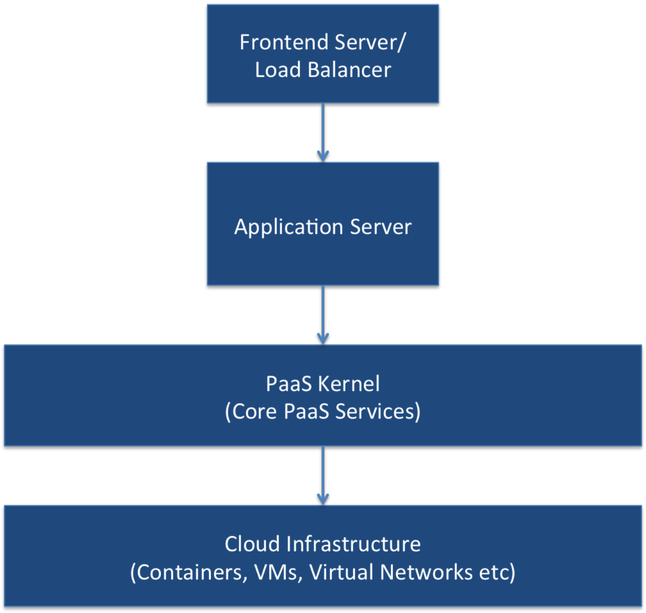

Figure 2.1: PaaS system organization.

issues.

PaaS clouds facilitate deploying and running applications that are directly consumed by human users and other client applications. As a result all the problems outlined in the previous chapter, such as poor development practices, lack of performance SLOs, and lack of performance debugging support directly impact PaaS clouds. Therefore PaaS clouds are ideal candidates for implementing the type of governance systems proposed in this work.

2.2.1 PaaS Architecture

Figure 2.1 shows the key layers of a typical PaaS cloud. Arrows indicate the flow of data and control in response to application requests. At the lowest level of a PaaS cloud is an infrastructure that consists of the necessary compute, storage and networking resources. How this infrastructure is set up may vary from a simple cluster of physical machines to a comprehensive Infrastructure-as-a-Service (IaaS) cloud. In large scale PaaS clouds, this layer typically consists of many virtual machines and/or containers with the ability to acquire more resources on the fly.

On top of the infrastructure layer lies the PaaS kernel – a collection of managed, scalable services that high-level application developers can compose into their applications. The provided kernel services may include database services, caching services, queuing services and more. The implementations of the kernel services are highly scalable, highly available (have SLOs associated with them), and automatically managed by the platform while being completely opaque to the application developers. Some PaaS clouds also provide a managed set of programming APIs (a "software development kit" or SDK) for the application developer to access these kernel services. In that case all interactions between the applications and the PaaS kernel must take place through the cloud provider specified SDK (e.g. Google App Engine [4], Microsoft Azure [33]).

One level above the PaaS kernel reside the application servers that are used to deploy and run applications. Application servers provide the necessary integration (linkage) between application code and the PaaS kernel services, while sandboxing application code for secure, multi-tenant execution. They also enable horizontal scaling of applications by running the same application on multiple application server instances.

The front-end and load balancing layer resides on top of the application servers layer. This layer is responsible for receiving all application requests, filtering them, and routing them to an appropriate application server instance for further execution. Front-end server is therefore the entry point for PaaS-hosted applications for all application clients.

Each of the above layers can span multiple processes, running over multiple physical or virtual machines. Therefore processing a single application request typically involves cooperation of multiple distributed processes and/or machines.

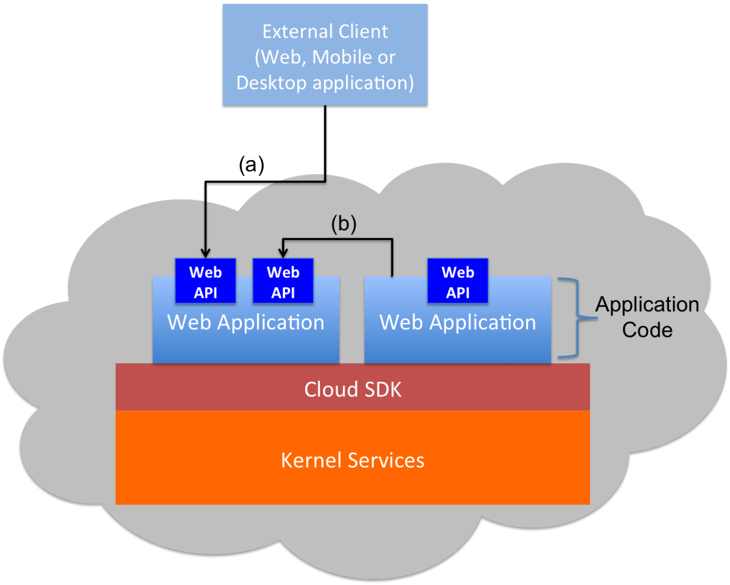

Figure 2.2: Applications deployed in a PaaS cloud: (a) An external client making requests to an application via the web API; (b) A PaaS-hosted application invoking another in the same cloud.

2.2.2 PaaS Usage Model

Three types of users interact with PaaS clouds.

Cloud administrators These are the personnel responsible for installing and maintaining the cloud platform software. They are always affiliated with the cloud platform provider.

Application developers These are the users who develop applications, and deploy them in the PaaS cloud.

Application clients These are the users that consume the applications deployed in a PaaS cloud. These include human users as well as other client applications that programmatically access PaaS-hosted applications.

Depending on how a particular PaaS cloud is set up (e.g. private or public cloud), the above three user groups may belong to the same or multiple organizations.

Figure 2.2 illustrates how the application developers interact with PaaS clouds. The cloud platform provides a set of kernel services. The PaaS SDK provides well defined interfaces (entry points) for these kernel services. The application developer uses the kernel services via the SDK to implement his/her application logic, and packages it as a web application. Developers then upload their applications to the cloud for deployment. Once deployed, the applications and any web APIs exported by them can be accessed via HTTP/S requests by external or co-located clients.

PaaS-hosted applications are typically developed and tested outside the cloud (on a developer's workstation), and then later uploaded to the cloud. Therefore PaaS-hosted applications typically undergo three phases during their life-cycle:

Development-time The application is being developed and tested on a developer's workstation

Deployment-time The finished application is being uploaded to the PaaS cloud for deployment

Run-time Application is running, and processing user requests

We explore ways to use these different phases to our advantage in order to minimize the governance overhead on running applications.

We use PaaS clouds in our research extensively both as case studies and experimental platforms. Specifically, we use Google App Engine and AppScale as test environments to experiment with our new governance systems. App Engine is a highly scalable public PaaS cloud hosted and managed by Google in their data centers. While it is open for anyone to deploy and run web applications, it is not open source software, and its internal deployment details are not commonly known. AppScale is open source software that can be used to set up a private cloud platform on one's own physical or virtual hardware.

AppScale is API compatible with App Engine (i.e. it supports the same cloud SDK), and hence any web application developed for App Engine can be deployed on AppScale without any code changes. In our experiments, we typically deploy AppScale over a small cluster of physical machines, or over a set of virtual machines provided by an IaaS cloud such as Eucalyptus.

By experimenting with real world PaaS clouds we demonstrate the practical feasibility and the effectiveness of the systems we design and implement. Furthermore, there are currently over a million applications deployed in App Engine, with a significant proportion of them being open source applications. Therefore we have access to a large number of real world PaaS applications to experiment with.

2.3 Governance

2.3.1 IT and SOA Governance

Traditionally, information and technology (IT) governance [15] has been a branch of corporate governance, focused on improving performance and managing the risks associated with the use of IT. A number of frameworks, models and even certification systems have emerged over time to help organizations implement IT governance [34, 35]. The primary goals of IT governance are three fold.

- Assure that the use of IT generates business value
- Oversee performance of IT usage and management
- Mitigate the risks of using IT

When the software engineering community started gravitating towards web services and service-oriented computing (SOC) [36, 37, 38], a new type of digital assets rose to prominence within corporate IT infrastructures – “services”. A service is a self-contained entity that logically represents a business activity (a functionality; e.g. user authentication, billing, VM management) while hiding its internal implementation details from the consumers [37]. Compositions of loosely-coupled, reusable, modular services soon replaced large monolithic software installations.

Services required new forms of governance for managing their performance and risks, and hence the notion of service-oriented architecture (SOA) governance came into existence [16, 17]. Multiple definitions of SOA governance are in circulation, but most of them agree that the purpose of SOA governance is to exercise control over services and associated processes (service development, testing, monitoring etc). A commonly used definition of SOA governance is ensuring and validating that service artifacts within the architecture are operating as expected, and maintaining a certain level of quality [16]. Consequently, a number of tools that help organizations implement SOA governance have also evolved [18, 20, 39, 19]. Since web services are the most widely used form of services in SOA-driven systems, most of these SOA governance tools have a strong focus on controlling web services [40].

Policies play a crucial role in all forms of governance. A policy is a specification of the acceptable behavior and the life cycle of some entity. The entity could be a department, a software system, a service or a human process such as developing a new application. In SOA governance, policies state how services should be developed, how they are to be deployed, how to secure them, and what level of quality of service to maintain while a service is in operation. SOA governance tools enable administrators to specify acceptable service behavior and life cycle as policies, and a software policy enforcement agent automatically enacts those policies to control various aspects of the services [41, 42, 43].

2.3.2 Governance for Cloud-hosted Applications

Cloud computing can be thought of as a heightened version of service-oriented computing. While classic SOC strives to offer data and application functionality as services, cloud computing offers a variety of computing resources as services, including hardware infrastructure (compute power, storage space and networking) and programming platforms. Moreover, the applications deployed on cloud platforms typically behave like services with separate implementation and interface components. Much like classic services, each cloud-hosted application can be a dependency for another co-located cloud application, or a client application running elsewhere (e.g. a mobile app).

Due to this resemblance, we argue that many concepts related to SOA governance are directly applicable to cloud platforms and cloud-hosted applications. We extend the definition of SOA governance, and define governance for cloud-hosted applications as the process of ensuring that the cloud-hosted applications operate as expected while maintaining a certain quality of service level.

Governance is a broad topic that allows room for many potential avenues of research. In our work we explore three specific features of governance as they apply to cloud-hosted applications.

Policy enforcement Policy enforcement refers to ensuring that all applications deployed in a cloud platform adhere to a set of policies specified by a cloud administrator. Some of these policies include specific dependency management practices, naming and packaging standards for software artifacts, software versioning requirements, and practices that enable software artifacts to evolve while maintaining backward compatibility. Others specify run-time constraints, which need to be enforced per application request.

Formulating performance SLOs This refers to automatic formulation of statistical bounds on the performance of cloud-hosted web applications. A service level objective (SLO) specifies a system's minimum quality of service (QoS) level in a measurable and controllable manner [44]. They may cover various QoS parameters such as availability, response time (latency), and throughput. A performance SLO specifies an upper bound on the application's response time, and the likelihood that bound is valid. Cloud administrators and application developers use performance SLOs to negotiate service level agreements (SLAs) with clients, and monitor applications for consistent operation. Clients use them to reason about the performance of downstream applications that depend on cloud-hosted applications.

Application performance monitoring Application performance monitoring (APM) refers to continuously monitoring cloud-hosted applications to detect violations of performance SLOs and other performance anomalies. It also includes diagnosing the root cause of each detected anomaly, thereby expediting remediation. This feature is useful for cloud administrators, application developers and clients alike.

None of the above features are implemented satisfactorily in the cloud technologies available today. In order to fill the gaps caused by these limitations, many third-party governance solutions that operate as external services have come into existence. For example, services like 3Scale [45], Apigee [46] and Layer7 [47] provide a wide range of access control and API management features for web applications served from cloud platforms. Similarly, services like New Relic [12], Dynatrace [13] and Datadog [14] provide monitoring support for cloud-hosted applications. But these services are expensive, and require additional programming and/or configuration. Some of them also require changes to applications in the form of code instrumentation. Moreover, since these services operate outside the cloud platforms they govern, they have limited visibility and control over the applications and related components residing in the cloud. A goal of our research is to facilitate governance from within the cloud, as an automated, cloud-native feature. We show that such built-in governance capabilities are more robust, effective and easy to use than external third-party solutions that overlay governance on top of the cloud.

2.3.3 API Governance

A cloud-hosted application is comprised of two parts – implementation and interface. The implementation contains the functionality of the application. It primarily consists of code that implements various application features. The interface, which abstracts and modularizes the implementation details of an application while making it network-accessible, is often referred to as a *web API* (or API in short). The API enables remote users and client applications to interact with the application by sending HTTP/S requests. The responses generated by an API could be based on HTML (for display on a web browser), or they could be based on a data format such as XML or JSON (for machine-to-machine interaction). Regardless of the technology used to implement an API, it is the part of the application that is visible to the remote clients.

Developers today increasingly depend on the functionality of already existing web applications in the cloud, which are accessible through their interfaces (APIs). Thus, a modern application often combines local program logic with calls to remote web APIs. This model significantly reduces both the programming and the maintenance workload associated with applications. In theory, because the APIs interface to software that is curated by cloud providers, the client application leverages greater scalability, performance, and availability in the implementations it calls upon through these APIs, than it would if those implementations were local to the client application (e.g. as locally available software libraries). Moreover, by accessing shared web applications, developers avoid "re-inventing the wheel" each time they need a commonly available application feature. The scale at which clouds operate ensures that the APIs can support the large volume of requests generated by the ever-growing client population.

As a result, web-accessible APIs and the software applications to which they provide access are rapidly proliferating. At the time of this writing, ProgrammableWeb [48], a popular web API index, lists more than $15,000$ publicly available web APIs, and a nearly 100% annual growth rate [49]. These APIs increasingly employ the REST (Representational State Transfer) architectural style [50], and many of them target commercial applications (e.g. advertising, shopping, travel, etc.). However, several non-commercial entities have also recently published web APIs, e.g. IEEE [51], UC Berkeley [52], and the US White House [53].

This proliferation of web APIs in the cloud demands new techniques that automate the maintenance and evolution of APIs as a first-class software resource – a notion that we refer to as *API governance* [54]. API management in the form of run-time mechanisms to implement access control is not new, and many good commercial offerings exist today [45, 46, 47]. However, API governance – consistent, generalized, policy implementation across multiple APIs in an administrative domain – is a new area of research made poignant by the emergence of cloud computing.

We design governance systems targeting the APIs exposed by the cloud-hosted web applications. We facilitate configuring and enforcing policies at the granularity of APIs. Similarly, we design systems that stipulate performance SLOs for individual APIs, and monitor them as separate independent entities.

Chapter 3

Governance of Cloud-hosted Applications Through Policy Enforcement

In this chapter we discuss implementing scalable, automated API governance through policy enforcement for cloud-hosted web applications. A lack of API governance can lead to many problems including security breaches, poor code reuse, violation of service-level objectives (SLOs), naming and branding issues, and abuse of digital assets by the API consumers. Unfortunately, most existing cloud platforms within which web APIs are hosted provide only minimal governance support; *e.g.* authentication and authorization. These features are important to policy implementation since governance often requires enforcement of access control on APIs. However, developers are still responsible for implementing governance policies that combine features such as API versioning, dependency management, and SLO enforcement as part of their respective applications.

Moreover, today's cloud platforms require that each application implements its own governance. There is no common, built-in system that enables cloud administrators to specify policies, which are automatically enforced on applications and their APIs. As a result, the application developers must be concerned with development issues (correct and efficient programming of application logic), as well as governance issues (administrative control and management) when implementing applications for the cloud.

Existing API management solutions [45, 46, 47] typically operate as external standalone services that are not integrated with the cloud. They do attempt to address governance concerns beyond mere access control. However, because they are not integrated within the cloud platform, their function is advisory and documentarian. That is, they do not possess the ability to implement full enforcement, and instead, alert operators to potential issues without preventing non-compliant behavior. They are also costly, and they can fail independently of the cloud, thereby affecting the scalability and availability of the software that they govern. Finally, it is not possible for them to implement policy enforcement at deployment-time – the phase of the software lifecycle during which an API change or a new API is being put into service. Because of the scale at which clouds operate, deployment-time enforcement is critical since it permits policy violations to be remediated before the changes are put into production (*i.e.* before run-time).

Thus, our thesis is that governance must be implemented as a built-in, native cloud service to overcome these shortcomings. That is, instead of an API management approach that layers governance features on top of the cloud, we propose to provide API governance as a fundamental service of the cloud platform. Cloud-native governance capabilities

- enable both deployment-time and run-time enforcement of governance policies as part of the cloud platform's core functionality,
- avoid inconsistencies and failure modes caused by integration and configuration of governance services that are not end-to-end integrated within the cloud fabric itself,
- leverage already-present cloud functionality such as fault tolerance, high availability

and elasticity to facilitate governance, and

- unify a vast diversity of API governance features across all stages of the API lifecycle (development, deployment, deprecation, retirement).

As a cloud-native functionality, such an approach also simplifies and automates the enforcement of API governance in the cloud. This in turns enables separation of governance concerns from development concerns for both cloud administrators as well as cloud application developers. The cloud administrators simply specify the policies, and trust the cloud platform to enforce them automatically on the applications. The application developers do not have to program any governance features into their applications, and instead rely on the cloud platform to perform the necessary governance checks either when the application is uploaded to the cloud, or when the application is being executed.

To explore the efficacy of cloud-integrated API governance, we have developed an experimental cloud platform that supports governance policy specification, and enforcement for the applications it hosts. EAGER – **E**nforced **A**PI **G**overnance **E**ngine for **R**EST – is a model and an architecture that is designed to be integrated within existing cloud platforms in order to facilitate API governance as a cloud-native feature. EAGER enforces proper versioning of APIs and supports dependency management and comprehensive policy enforcement at API deployment-time.

Using EAGER, we investigate the trade-offs between deployment-time policy enforcement and run-time policy enforcement. Deployment-time enforcement is attractive for several reasons. First, if only run-time API governance is implemented, policy violations will go undetected until the offending APIs are used, possibly in a deep stack or call path in an application. As a result, it may be difficult or time consuming to pinpoint the specific API and policy that are being violated (especially in a heavily loaded web service). In these settings, multiple deployments and rollbacks may occur before a policy violation is triggered making it difficult or impossible to determine the root cause of the violation. By enforcing governance as much as possible at deployment-time, EAGER implements "fail fast" in which violations are detected immediately making diagnosis and remediation less complex. Further, from a maintenance perspective, the overall system is prevented from entering a non-compliant state, which aids in the certification of regulatory compliance. In addition, run-time governance typically implies that each API call will be intercepted by a policy-checking engine that uses admission control, and an enforcement mechanism creating scalability concerns. Because deployment events occur before the application is executed, traffic need not be intercepted and checked "in flight", thus improving the scaling properties of governed APIs. However, not all governance policies can be implemented strictly at deployment-time. As such, EAGER includes run-time enforcement facilities as well. The goal of our research is to identify how to implement enforced API governance most efficiently by combining deployment-time enforcement where possible, and run-time enforcement where necessary.

EAGER implements policies governing the APIs that are deployed within a single administrative domain (i.e. a single cloud platform). It treats APIs as first-class software assets due to the following reasons.

- APIs are often longer lived than the individual clients that use them or the implementations of the services that they represent.
- APIs represent the "gateway" between software functionality consumption (API clients and users) and service production (web service implementation).

EAGER acknowledges the crucial role APIs play by separating the API life cycle management from that of the service implementations and the client users. It facilitates policy definition and enforcement at the API level, thereby permitting the service and client implementations to change independently without the loss of governance control.

EAGER further enhances software maintainability by guaranteeing that developers reuse existing APIs when possible to create new software artifacts (to prevent API redundancy and unverified API use). At the same time, it tracks changes made by developers to already deployed web APIs to prevent any backwards-incompatible API changes from being put into production.

EAGER includes a language for specifying API governance policies. The EAGER language is distinct from existing policy languages like WS-Policy [55, 56] in that it avoids the complexities of XML, and it incorporates a developer-friendly Python programming language syntax for specifying complex policy statements in a simple and intuitive manner. Moreover, we ensure that specifying the required policies is the only additional activity that API providers should perform in order to use EAGER. All other API governance related verification and enforcement work is carried out by the cloud platform automatically.

To evaluate the feasibility and performance of the proposed architecture, we prototype the EAGER concepts in an implementation that extends AppScale [57], an open source cloud platform that emulates Google App Engine [4]. We describe the implementation and integration as an investigation of the generality of the approach. By focusing on deployment actions and run-time message checking, we believe that the integration methodology will translate to other extant cloud platforms.

We further show that EAGER API governance and policy enforcement impose a negligible overhead on the application deployment process, and the overhead is linear in the number of APIs in the applications being validated. Finally, we show that EAGER is able to scale to tens of thousands of deployed web APIs and hundreds of governance policies.

In the sections that follow, we present some background on cloud-hosted APIs, and overview the design and implementation of EAGER. We then empirically evaluate EA-

GER using a wide range of APIs and experiments. Finally, we discuss related work, and conclude the chapter.

3.1 Enforcing API Governance in Cloud Settings

Software engineering best practices separate the service implementation from API, both during development and maintenance. The service implementation and API are integrated via a "web service stack" that implements functionality common to all web services (message routing, request authentication, etc.). Because the API is visible to external parties (*i.e.* clients of the services), any changes to the API impacts users and client applications not under the immediate administrative control of the API provider. For this reason, API features usually undergo long periods of "deprecation" so that independent clients of the services can have ample time to "migrate" to newer versions of an API. On the other hand, technological innovations often prompt service reimplementation and/or upgrade to achieve greater cost efficiencies, performance levels, etc. Thus, APIs typically have a more slowly evolving and longer lasting lifecycle than the service implementations to which they provide access.

Modern computing clouds, especially clouds implementing some form of Platform-as-a-Service (PaaS) [58], have accelerated the proliferation of web APIs and their use. Most PaaS clouds [57, 59, 8] include features designed to ease the development and hosting of web APIs for scalable use over the Internet. This phenomenon is making API governance an absolute necessity in cloud environments.

In particular, API governance promotes code reuse among developers since each API must be treated as a tracked and controlled software entity. It also ensures that software users benefit from change control since the APIs they depend on change in a controlled and non-disruptive manner. From a maintenance perspective, API governance makes it possible to enforce best-practice coding procedures, naming conventions, and deployment procedures uniformly. API governance is also critical to API lifecycle management – the management of deployed APIs in response to new feature requests, bug fixes, and organizational priorities. API "churn" that results from lifecycle management is a common phenomenon and a growing problem for web-based applications [60]. Without proper governance systems to manage the constant evolution of APIs, API providers run the risk of making their APIs unreliable while potentially breaking downstream applications that depend on the APIs.

Unfortunately, most web technologies used to develop and host web APIs do not provide API governance facilities. This missing functionality is especially glaring for cloud platforms that are focused on rapid deployment of APIs at scale. Commercial pressures frequently prioritize deployment speed and scale over longer-term maintenance considerations only to generate unanticipated future costs.

As a partial countermeasure, developers of cloud-hosted applications often undertake additional tasks associated with implementing custom *ad hoc* governance solutions using either locally developed mechanisms or loosely integrated third-party API management services. These add-on governance approaches often fall short in terms of their consistency and enforcement capabilities since by definition they have to operate outside the cloud (either external to it or as another cloud-hosted application). As such, they do not have the end-to-end access to all the metadata and cloud-internal control mechanisms that are necessary to implement strong governance at scale.

In a cloud setting, enforcement of governance policies on APIs is a tradeoff between what can be enforced, and when they are enforced. Performing policy enforcement at application run-time provides full control over what can be enforced, since the policy engine can intercept and control all operations and instructions executed by the application. However, this approach is highly intrusive, which introduces complexity and performance overhead. Alternatively, attempting to enforce policies prior to application's execution is attractive in terms of performance, but it necessarily limits what can be enforced. For example, ensuring that an application does not connect to a specific network address and/or port requires run-time traffic interception, typically by a firewall that is interposed between the application and the offending network. Enforcing such a policy can only be performed during run-time.

For policy implementation, often the additional complexities and overhead introduced by run-time enforcement outweigh its benefits. For example, in an application that consists of API calls to services that, in turn, make calls to other services, run-time policy enforcement can make violations difficult to resolve, especially when the interaction between services is non-deterministic. When a specific violation occurs, it may be "buried" in a lattice of API invocations that is complex to traverse, especially if the application itself is designed to handle large-scale request traffic loads.

Ideally, then, enforcement takes place as non-intrusively as possible before the application begins executing. In this way, a violation can be detected and resolved *before the API is used*, thereby avoiding possible degradations in user-experience that run-time checks and violations may introduce. The drawback of attempting to enforce all governance before the application begins executing is that policies that express restrictions only resolvable at run time cannot be implemented. Thus, for scalable applications that use API calls internally in a cloud setting, an API governance approach should attempt to implement as much as possible no later than deployment time, but must also include some form of run-time enforcement.

Note that the most effective approach to implementing a specific policy may not always be clear. For example, user authentication is usually implemented as a run-time policy check for web services since users enter and leave the system dynamically. However it is possible to check statically, at deployment time, whether the application is consulting with a specific identity management service (accessed by a versioned API) thereby enabling some deployment-time enforcement.

Thus, any efficient API governance solution for clouds must include the following functionalities.

- **Policy Specification Language** – The system must include a way to specify policies that can be enforced either at deployment-time (or sooner) or, ultimately at run-time.
- **API Specification Language** – Policies must be able to refer to API functionalities to be able to express governance edicts for specific APIs or classes of APIs.
- **Deployment-time Control** – The system must be able to check policies no later than the time that an application is deployed.
- **Run-time Control** – For policies that cannot be enforced before runtime, the system must be able to intervene dynamically.

In addition, a good solution should automate as much of the implementation of API governance as possible. Automation in a cloud context serves two purposes. First, it enables scale by allowing potentially complex optimizations to be implemented reliably by the system, and not by manual intervention. Secondly, automation improves repeatability and auditability thereby ensuring greater system integrity.

3.2 EAGER

To experiment with API governance in cloud environments, we devise EAGER – an architecture for implementing governance that is suitable for integration as a cloud-native feature. EAGER leverages existing SOA governance techniques and best practices,

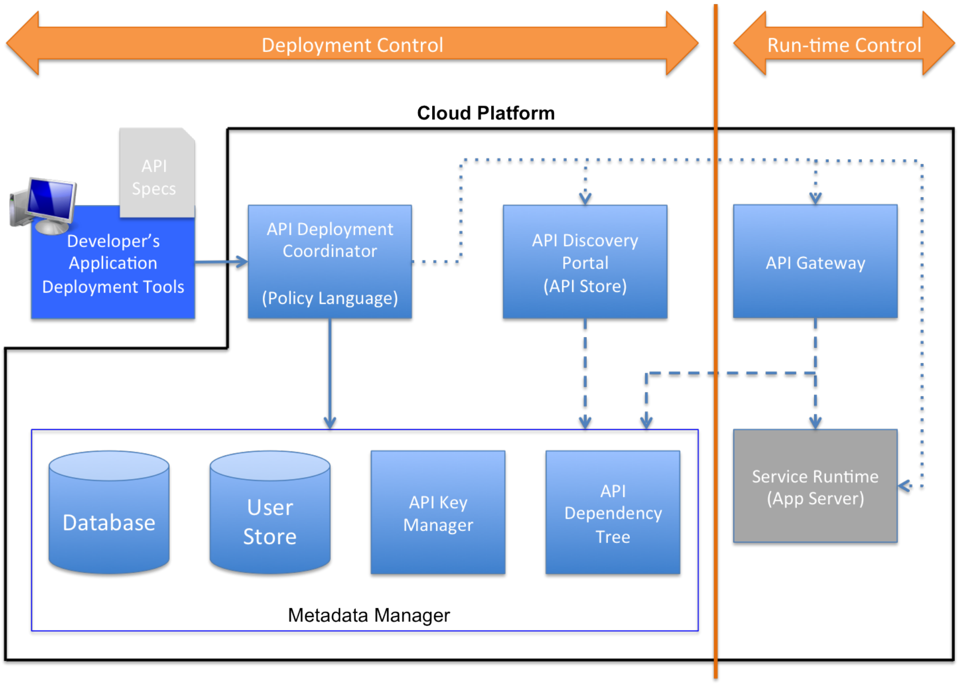
Deployment Control
Run-time Control
Cloud Platform
API Specs
Developer's Application Deployment Tools
API Deployment Coordinator (Policy Language)
API Discovery Portal (API Store)
API Gateway
Database
User Store
API Key Manager
API Dependency Tree
Metadata Manager
Service Runtime (App Server)

Figure 3.1: EAGER Architecture

and adapts them to make them suitable for cloud platform-level integration. In this section, we overview the high-level design of EAGER, its main components, and the policy language. Our design is motivated by two objectives. First, we wish to verify that the integration among policy specification, API specification, deployment-time control, and run-time control is feasible in a cloud setting. Secondly, we wish to use the design as the basis for a prototype implementation that we could use to evaluate the impact of API governance empirically.

EAGER is designed to be integrated with PaaS clouds. PaaS clouds accept code that is then deployed within the platform so that it may make calls to kernel services offered by the cloud platform, or other applications already deployed in the cloud platform via their APIs. EAGER intercepts all events related to application deployment within the cloud, and enforces governance checks at deployment-time. When a policy verification check fails, EAGER aborts the deployment of the application, and logs the information necessary to perform remediation. EAGER assumes that it is integrated with the cloud, and that the cloud initiates in a policy compliant state (*i.e.* there are no policy violations when the cloud is first launched before any applications are deployed). We use the term "initiates" to differentiate the first clean launch of the cloud, from a platform restart. EAGER must be able to maintain compliance across restarts, but it assumes that when the cloud is first installed and suitably tested, it is in a policy compliant state. Moreover, it maintains the cloud in a policy compliant state at all times. That is, with EAGER active, the cloud is automatically prevented from transitioning out of policy compliance due to a change in the applications it hosts.

Figure 3.1 illustrates the main components of EAGER (in blue), and their interactions. Solid arrows represent the interactions that take place during application deployment-time, before an application has been validated for deployment. Short-dashed arrows indicate the interactions that take place during deployment-time, after an application has been successfully validated. Long-dashed arrows indicate interactions at run-time. The diagram also outlines the components of EAGER that are used to provide deployment-time control and run-time control. Note that some components participate in interactions related to both deployment and run-time control (e.g. metadata manager).

EAGER is invoked by the cloud whenever a user attempts to deploy an application in the cloud. The cloud's application deployment mechanisms must be altered so that each deployment request is intercepted by EAGER, which then performs the required governance checks. If a governance check fails, EAGER preempts the application deployment, logs relevant data pertaining to the event for later analysis, and returns an error. Otherwise, it proceeds with the application deployment by activating the deployment mechanisms on the user's behalf.

Architecturally, the deployment action requires three inputs: the policy specification governing the deployment, the application code to be deployed, and a specification of the APIs that the application exports. EAGER assumes that cloud administrators have developed and installed policies (stored in the metadata manager) that are to be checked against all deployments. API specifications for the application must also be available to the governance framework. Because the API specifications are to be derived from the code (and are, thus, under developer control and not administrator control) our design assumes that automated tools are available to perform analysis on the application, and generate API specifications in a suitable API specification language. These specifications must be present when the deployment request is considered by the platform. In the prototype implementation described in section 3.3, the API specifications are generated as part of the application development process (*e.g.* by the build system). They may also be offered as a trusted service hosted in the cloud. In this case, developers will submit their source code to this service, which will generate the necessary API specifications in the cloud, and trigger the application deployment process via EAGER.

The proposed architecture does not require major changes to the existing components of the cloud, since its deployment mechanisms are likely to be web service based. However, EAGER does require integration at the platform level. That is, it must be a trusted component in the cloud platform.

3.2.1 Metadata Manager

The metadata manager stores all the API metadata in EAGER. This metadata includes policy specifications, API names, versions, specifications and dependencies. It uses the dependency information to compute the dependency tree among all deployed APIs and applications. Additionally, the metadata manager also keeps track of developers, their subscriptions to various APIs, and the access credentials (API keys) issued to them. For these purposes, the metadata manager must logically include both a database, and an identity management system.

The metadata manager is exposed to other components through a well defined web service interface. This interface allows querying existing API metadata and updating them. In the proposed model, the stored metadata is updated occasionally – only when a new application is deployed or when a developer subscribes to a published API. Therefore the Metadata Manager does not need to support a very high write throughput. This performance characteristic allows the Metadata Manager to be implemented with strong transactional semantics, which reduces the development overhead of other components that rely on metadata manager. Availability can be improved via simple replication methods.

3.2.2 API Deployment Coordinator

The API Deployment Coordinator (ADC) intercepts all application deployment requests, and determines whether they are suitable for deployment, based on a set of policies specified by the cloud administrators. It receives application deployment requests via a web service interface. At a high-level, ADC is the most important entity in the EAGER's deployment-time control strategy.

An application deployment request contains the name of the application, version number, names and versions of the APIs exported by the application, detailed API specifications, and other API dependencies as declared by the developer. Application developers only need to specify explicitly the name and version of the application and the list of dependencies (i.e. APIs consumed by the application). All other metadata can be computed automatically by performing introspection on the application source code.

The API specifications used to describe the web APIs should state the operations and the schema of their inputs and outputs. Any standard API description language can be used for this purpose, as long as it clearly describes the schema of the requests and responses. For describing REST interfaces, we can use Web Application Description Language (WADL) [61], Swagger [62], RESTful API Modeling Language (RAML) or any other language that provides similar functionality.

When a new deployment request is received, the ADC checks whether the application declares any API dependencies. If so, it queries the metadata manager to make sure that all the declared dependencies are already available in the cloud. Then it inspects the enclosed application metadata to see if the current application exports any web APIs. If the application exports at least one API, the ADC makes another call to the metadata manager, and retrieves any existing metadata related to that API. If the metadata manager cannot locate any data related to the API in question, ADC assumes it to be a brand new API (i.e. no previous version of that API has been deployed in the cloud), and proceeds to the next step of the governance check, which is policy validation. However, if any metadata regarding the API is found, then the ADC is dealing with an API update. In this case, the ADC compares the old API specifications with the latest ones provided in the application deployment request to see if they are compatible.

To perform this API compatibility verification, the ADC checks to see whether the latest specification of an API contains all the operations available in the old specification. If the latest API specification is missing at least one operation that it had previously, the ADC reports this to the user and aborts the deployment. If all the past operations are present in the latest specification, the ADC performs a type check to make sure that all past and present operations are type compatible. This is done by performing recursive introspection on the input and output data types declared in the API specifications. EAGER looks for type compatibility based on the following rules inspired by Hoare logic [63], and the rules of type inheritance from object oriented programming.

- New version of an input type is compatible with the old version of an input type, if the new version contains either all or less attributes than the old version, and any new attributes that are unique to the new version are optional.
- New version of an output type is compatible with the old version of an output type, if the new version contains either all or more attributes than the old version.

In addition to the type checks, ADC may also compare other parameters declared in the API specifications such as HTTP methods, mime types and URL patterns. We have also explored and published results on using a combination of syntactic and semantic comparison to determine the compatibility between APIs [60, 64]. Once the API specifications have been successfully compared without error, and the compatibility established, the ADC initiates policy validation.

3.2.3 EAGER Policy Language and Examples

Policies are specified by cloud or organizational administrators using a subset of the popular Python programming language. This design choice is motivated by several reasons.

- A high-level programming language such as Python is easier to learn and use for policy implementors.
- Platform implementors can use existing Python interpreters to parse and execute policy files. Similarly, policy implementors can use existing Python development tools to write and test policies.
- In comparison to declarative policy languages (e.g. WS-Policy), a programming language like Python offers more flexibility and expressive power. For example, a policy may perform some local computation, and use the results in its enforcement

clauses. The control flow tools of the language (e.g. conditionals, loops) facilitate specifying complex policies.

- The expressive power of the language can be closely regulated by controlling the set of allowed built-in modules and functions.

We restrict the language to prevent state from being preserved across policy validations. In particular, the EAGER policy interpreter disables file and network operations, third party library calls, and other language features that allow state to persist across invocations. In addition, EAGER processes each policy independently of others (i.e. each policy must be self-contained and access no external state). All other language constructs and language features can be used to specify policies in EAGER.

To accommodate built-in language APIs that the administrators trust by *fiat*, all module and function restrictions of the EAGER policy language are enforced through a configurable white-list. The policy engine evaluates each module and function reference found in policy specifications against this white-list to determine whether they are allowed in the context of EAGER. Cloud administrators have the freedom to expand the set of allowed built-in and third party modules by making changes to this white-list.

As part of policy language, EAGER defines a set of assertions that policy writers can use to specify various checks to perform on the applications. Listing 3.1 shows the assertions currently supported by EAGER.

Listing 3.1: Assertions supported by the EAGER policy language.

```
assert_true(condition, optional_error_msg)
assert_false(condition, optional_error_msg)
assert_app_dependency(app, d_name, d_version)
assert_not_app_dependency(app, d_name, d_version)
assert_app_dependency_in_range(app, name,\
  lower, upper, exclude_lower, exclude_upper)
```

In addition to these assertions, EAGER adds a function called “compare_versions” to the list of available built-in functions. Policy implementors can use this function to compare version number strings associated with applications and APIs.

In the remainder of this section we illustrate the use of the policy language through several examples. The first example policy, shown in listing 3.2, mandates that any application or mash-up that uses both Geo and Direction APIs must adhere to certain versioning rules. More specifically, if the application uses Geo 3.0 or higher, it must use Direction 4.0 or higher. Note that the version numbers are compared using the “compare_versions” functions described earlier.

Listing 3.2: Enforcing API version comparison

```
g = filter(lambda dep: dep.name == ‘Geo’, app.dependencies)
d = filter(lambda dep: dep.name == ‘Direction’, app.dependencies)
if g and d:
  g_api, d_api = g[0], d[0]
  if compare_versions(g_api.version, ‘3.0’) >= 0:
    assert_true(compare_versions(d_api.version, ‘4.0’) >= 0)
```

In listing 3.2, *app* is a special immutable logical variable available to all policy files. This variable allows policies to access information pertaining to the current application deployment request. The assert_true and assert_false functions allow testing for arbitrary conditions, thus greatly improving the expressive power of the policy language.

Listing 3.3 shows a policy file that mandates that all applications deployed by the “admin@test.com” user must have role-based authentication enabled, so that only users in the “manager” role can access them. To carry out this check the policy accesses the security configuration specified in the application descriptor (e.g. the web.xml for a Java application).

Listing 3.3: Enforcing role-based authorization.

```
if app.owner == ‘admin@test.com’:
  roles = app.web_xml[‘security-role’]
  constraints = app.web_xml[‘security-constraint’]
  assert_true(roles and constraints)
  assert_true(len(roles) == 1)
  assert_true(‘manager’ == roles[0][‘role-name’])
```

Listing 3.4 shows an example policy, which mandates that all deployed APIs must explicitly declare an operation which is accessible through the HTTP OPTIONS method. This policy further ensures that these operations return a description of the API in the Swagger [62] machine-readable API description language.

Listing 3.4: Enforcing APIs to publish a description.

```
options = filter(lambda op : op.method == ‘OPTIONS’,
  api.operations)
assert_true(options, ‘API does not support OPTIONS’)
assert_true(options[0].type == ‘swagger.API’,
  ‘Does not return a Swagger description’)
```

Returning machine-readable API descriptions from web APIs makes it easier to automate the API discovery and consumption processes. Several other research efforts confirm the need for such descriptions [65, 66]. A policy such as this can help enforce such practices, thus resulting in a high-quality API ecosystem in the target cloud.

The policy above also shows the use of the second and optional string argument to the assert_true function (the same is supported by assert_false as well). This argument can be used to specify a custom error message that will be returned to the application developer, if his/her application violates the assertion in question.

The next example policy prevents developers from introducing dependencies on deprecated web APIs. Deprecated APIs are those that have been flagged by their respective authors for removal in the near future. Therefore introducing dependencies on such APIs is not recommended. The policy in listing 3.5 enforces this condition in the cloud.

Listing 3.5: Preventing dependencies on deprecated APIs.

```
deprecated = filter(
  lambda dep : dep.status == ’DEPRECATED’,
  app.dependencies)
assert_false(deprecated,
  ’Must not use a deprecated dependency’)
```

Listing 3.6: Tenant-aware policy enforcement.

```
if app.owner.endswith(‘@engineering.test.com’):
  assert_app_dependency(app, ‘Log’, ‘1.0’)
elif app.owner.endswith(‘@sales.test.com’):
  assert_app_dependency(app, ‘AnalyticsLog’, ‘1.0’)
else:
  assert_app_dependency(app, ‘GenericLog’, ‘1.0’)
```

Our next example presents a policy that enforces governance rules in a user-aware (i.e. tenant-aware) manner. Assume a multi-tenant private PaaS cloud that is being used by members of the development team and the sales team of a company. The primary goal in this case is to ensure that applications deployed by both teams log their activities using a set of preexisting logging APIs. However, we further want to ensure that applications deployed by the sales team log their activities using a special analytics API. A policy such as the one in listing 3.6 can enforce these conditions.

The example in listing 3.7 shows a policy, which mandates that all HTTP GET operations exposed by APIs must support paging. APIs that do so define two input parameters named “start” and “count” to the GET call.

Listing 3.7: Enforcement of paging functionality in APIs.

```
for api in app.api_list:
  get = filter(lambda op : op.method == ‘GET’,
    api.operations)
  for op in get:
    param_names = map(lambda p : p.name,
      op.parameters)
    assert_true(‘start’ in param_names and
      ‘count’ in param_names)
```

This policy accesses the metadata of API operations that is available in the API descriptions. Since API descriptions are auto-generated from the source code of the APIs, this policy indirectly references information pertaining to the actual API implementations.

Finally, we present an example for the HTTP POST method. The policy in listing 3.8 mandates that all POST operations exposed by an API are secured with OAuth version 2.0.

Listing 3.8: Enforcement of OAuth-based authentication for APIs.

```
for api in app.api_list:
  post = filter(lambda op : op.method == ‘POST’,
        api.operations)
  for op in post:
    assert_true(op.authorizations.get(‘oauth2’))
```

EAGER places no restrictions on how many policy files are specified by administrators. Applications are validated against each policy file. Failure of any assertion in any policy file causes the ADC to abort application deployment. Once an application is checked against all applicable policies, ADC persists the latest application and API metadata into the Metadata Manager. At this point, the ADC may report success to the user, and proceed with application deployment. In a PaaS setting this deployment activity typically involves three steps:

1. Deploy the application in the cloud application run-time (application server).
2. Publish the APIs enclosed in the application and their specifications to the API Discovery Portal or catalog.
3. Publish the APIs enclosed in the application to an API Gateway server.

Step 1 is required to complete the application deployment in the cloud even without EAGER. We explain the significance of steps 2 and 3 in the following subsections.

3.2.4 API Discovery Portal

The API Discovery Portal (ADP) is an online catalog where developers can browse available web APIs. Whenever the ADC approves and deploys a new application, it registers all the APIs exported by the application in ADP. EAGER mandates that any developer interested in using an API, first subscribe to that API and obtain the proper credentials (API keys) from the ADP. The API keys issued by the ADP can consist of an OAuth [67] access token (as is typical of many commercial REST-based web services) or a similar authorization credential, which can be used to identify the developer/application that is invoking the API. This credential validation process is used for auditing, and run-time governance in EAGER.

The API keys issued by the ADP are stored in the metadata manager. When a programmer develops a new application using one or more API dependencies, we require the developer to declare its dependencies along with the API keys obtained from the ADP. The ADC verifies this information against the metadata manager as a part of its dependency check, and ensures that the declared dependencies are correct and the specified API keys are valid.

Deployment-time governance policies may further incentivize the declaration of API dependencies explicitly by making it impossible to call an API without first declaring it as a dependency along with the proper API keys. These types of policies can be implemented with minor changes to the application run-time in the cloud so that it loads the API credentials from the dependency declaration provided by the application developer.

In addition to API discovery, the ADP also provides a user interface for API authors to select their own APIs and deprecate them or retire them. Deprecated APIs will be removed from the API search results of the portal, and application developers will no longer be able to subscribe to them. However, already existing subscriptions and API keys will continue to work until the API is eventually retired. The deprecation is considered a courtesy notice for application developers who have developed applications using the API, to migrate their code to a newer version of the API. Once retired, any applications that have not still been migrated to the latest version of the API will cease to operate.

3.2.5 API Gateway

Run-time governance of web services by systems such as Synapse [68] make use of an API “proxy” or gateway. The EAGER API gateway does so to intercept API calls and validate the API keys contained within them. EAGER intercepts requests by blocking direct access to the APIs in the application run-time (app servers), and publishing the API Gateway address as the API endpoint in the ADP. We do so via firewall rules that prevent the cloud app servers from receiving any API traffic from a source other than the API gateway. Once the API gateway validates an API call, it routes the message to the application server in the cloud platform that hosts the API.

The API gateway can be implemented via one or more load-balanced servers. In addition to API key validation, the API gateway can perform other functions such as monitoring, throttling (rate limiting), and run-time policy validation.

3.3 Prototype Implementation

We implement a prototype of EAGER by extending AppScale [57], an open source PaaS cloud that is functionally equivalent to Google App Engine (GAE). AppScale supports web applications written in Python, Java, Go and PHP. Our prototype implements governance for all applications and APIs hosted in an AppScale cloud.

As described in subsection 3.2.3, EAGER's policy specification language is based on Python. This allows the API deployment coordinator (also written in Python) to execute the policies directly using a modified Python interpreter to implement the restrictions previously discussed.

The prototype relies on a separate tool chain (*i.e.* one not hosted as a service in the cloud) to automatically generate API specifications and other metadata (*c.f.* Section 3.2.2), which currently supports only the Java language. Developers must document the APIs manually for web applications implemented in languages other than Java.

Like most PaaS technologies, AppScale includes an application deployment service that distributes, launches and exports an application as a web-accessible service. EAGER controls this deployment process according to the policies that the platform administrator specifies.

3.3.1 Auto-generation of API Specifications

To auto-generate API specifications, the build process of an application must include an analysis phase that generates specifications from the source code. Our prototype includes two stand-alone tools for implementing this “build-and-analyze” function.

1. An Apache Maven archetype that is used to initialize a Java web application project, and

2. A Java doclet that is used to auto-generate API specifications from web APIs implemented in Java

Developers invoke the Maven archetype from the command-line to initialize a new Java web application project. Our archetype sets up projects with the required AppScale (GAE) libraries, Java JAX-RS [69] (Java API for RESTful Web Services) libraries, and a build configuration.

Once the developer creates a new project using the archetype, he/she can develop web APIs using the popular JAX-RS library. When the code is developed, it can be built using our auto-generated Maven build configuration, which introspects the project source code to generate specifications for all enclosed web APIs using the Swagger [70] API description language. It then packages the compiled code, required libraries, generated API specifications, and the dependency declaration file into a single, deployable artifact.

Finally, the developer submits the generated artifact for deployment to the cloud platform, which in our prototype is done via AppScale developer tools. To enable this, we modify the tools so that they send the application deployment request to the EAGER ADC and delegate the application deployment process to EAGER. This change required just under 50 additional lines of code in AppScale.

EAGER Component	Implementation Technology
Metadata Manager	MySQL
API Deployment Coordinator	Native Python implementation
API Discovery Portal	WSO2 API Manager [71]
API Gateway	WSO2 API Manager

Table 3.1: Implementation technologies used to implement the EAGER prototype

3.3.2 Implementing the Prototype

Table 3.1 lists the key technologies that we use to implement various EAGER functionalities described in section 3.2 as services within AppScale. For example, AppScale controls the lifecycle of the MySQL database as it would any of its other constituent services. EAGER incorporates the WSO2 API Manager [72] for use as an API discovery mechanism, and to implement any run-time policy enforcement. In the prototype, the API gateway does not share policies expressed in the policy language with the ADC. This integration is left to be implemented in the future.

Also, according to the architecture of EAGER, metadata manager is the most suitable location for storing all policy files. The ADC may retrieve the policies from the metadata manager through its web service interface. However, for simplicity, our current prototype stores the policy files in a file system, that the ADC can directly read from. In a more sophisticated future implementation of EAGER, we will move all policy files to the metadata manager where they can be better managed.

3.4 Experimental Results

In this section, we describe our empirical evaluation of the EAGER prototype, and evaluate its overhead and scaling characteristics. To do so, we populate the EAGER database (metadata manager) with a set of APIs, and then examine the overhead associated with governing a set of sample AppScale applications (shown in Table 3.2) for varying degrees of policy specifications and dependencies. In the first set of results we use randomly generated APIs, so that we may vary different parameters that may affect performance. We then follow with a similar analysis using a large set of API specifications "scraped" from the ProgrammableWeb [48] public API registry.

Note that all the figures included in this section present the average values calculated over three sample runs. The error bars cover an interval of two standard deviations centered at the calculated sample average.

We start by presenting the time required for AppScale application deployment without EAGER, as it is this process on which we piggyback EAGER support. These measurements are conservative since they are taken from a single node deployment of AppScale where there is no network communication overhead. Our test AppScale cloud is deployed on an Ubuntu 12.04 Linux virtual machine with a 2.7 GHz CPU, and 4 GB of memory. In practice AppScale is deployed over multiple hosts in a distributed manner where different components of the cloud platform must communicate via the network.

Table 3.2 lists a number of App Engine applications that we consider, their artifact size, and their average deployment times across three runs, on AppScale without EAGER. We also identify the number of APIs and dependencies for each application in the `Description` column. These applications represent a wide range of programming languages, application sizes, and business domains.

On average, deployment without EAGER takes 34.5 seconds, and this time is correlated with application artifact size. The total time consists of network transfer time of the application to the cloud (which in this case is via localhost networking), and disk copy time to the application servers. For actual deployments, both components are likely to increase due to network latency, available bandwidth, contention, and large numbers of distributed application servers.

Application	Description	Size (MB)	Deployment Time (s)
guestbook-py	A simple Python web application that allows users to post comments and view them	0.16	22.13
guestbook-java	A Java clone of the guestbook-python app	52	24.18
appinventor	A popular open source web application that enables creating mobile apps	198	111.47
coursebuilder	A popular open source web application used to facilitate teaching online courses	37	23.75
hawkeye	A sample Java application used to test AppScale	35	23.37
simple-jaxrs-app	A sample JAXRS app that exports 2 web APIs	34	23.45
dep-jaxrs-app	A sample JAXRS app that exports a web API and has one dependency	34	23.72
dep-jaxrs-app-v2	A sample JAXRS app that exports 2 web APIs and has one dependency	34	23.95

Table 3.2: Sample AppScale applications

3.4.1 Baseline EAGER Overhead by Application

Figure 3.2 shows the average time in seconds taken by EAGER to validate and verify each application. We record these results on an EAGER deployment without any policies deployed, and without any prior metadata recorded in the metadata manager (that is, an unpopulated database of APIs). We present the values as absolute measurements (here and henceforth) because of the significant difference between them and deployment times on AppScale without EAGER (100's of milliseconds compared to 10's of seconds). We can alternatively observe this overhead as a percentage of AppScale deployment time by dividing these times by those shown in Table 3.2.

Note that some applications do not export any web APIs. For these EAGER over-

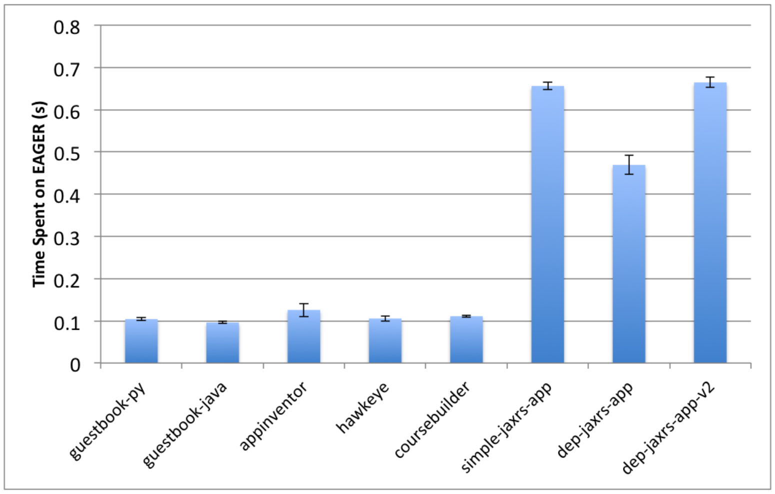

Figure 3.2: Absolute mean overhead of EAGER by application. Each data point averages three executions, the error bars are two standard deviations, and the units are seconds.

head is negligibly small (approximately 0.1s). This result indicates that EAGER does not impact deployment time of applications that do not require API governance. For applications that do export web APIs, the recorded overhead measurements include the time to retrieve old API specifications from the metadata manager, the time to compare the new API specifications with the old ones, the time to update the API specifications and other metadata in the Metadata Manager, and the time to publish the updated APIs to the cloud. The worst case observed overhead for governed APIs (simple-jaxrs-app in the figure 3.2) is 2.8%.

3.4.2 Impact of Number of APIs and Dependencies

Figure 3.3 shows that EAGER overhead grows linearly with the number of APIs exported by an application. This scaling occurs because the current prototype implementation iterates through the APIs in the application sequentially, and records the API metadata in the metadata manager. Then EAGER publishes each API to the ADP and API Gateway. This sequencing of individual EAGER events, each of which generates a

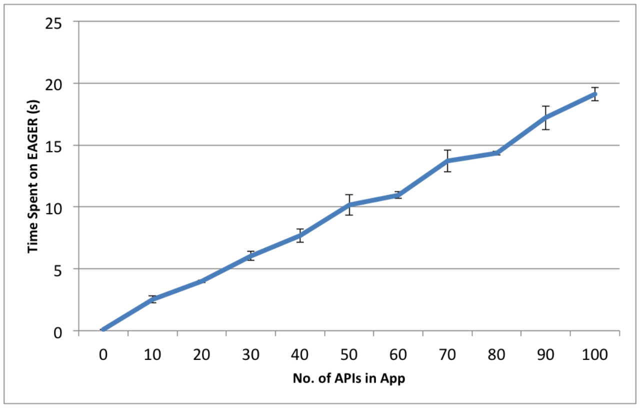

Figure 3.3: Average EAGER overhead vs. number of APIs exported by the application. Each data point averages three executions, the error bars are two standard deviations, and the units are seconds.

separate web service call, represents an optimization opportunity via parallelization in future implementations.

At present we expect most applications deployed in cloud to have a small to moderate number of APIs (10 or fewer). With this API density EAGER's current scaling is adequate. Even in the unlikely case that a single application exports as many as 100 APIs, the average total time for EAGER is under 20 seconds.

Next, we analyze EAGER overhead as the number of dependencies declared in an application grows. For this experiment, we first populate the EAGER metadata manager with metadata for 100 randomly generated APIs. To generate random APIs we use the API specification auto-generation tool to generate fictitious APIs with randomly varying numbers of input/output parameters. Then we deploy an application on EAGER which exports a single API, and declares artificial dependencies on the set of fictitious APIs that are already stored in the Metadata Manager. We vary the number of declared dependencies and observe the EAGER overhead.

Figure 3.4 shows the results of these experiments. EAGER overhead does not appear to be significantly influenced by the number of dependencies declared in an application.

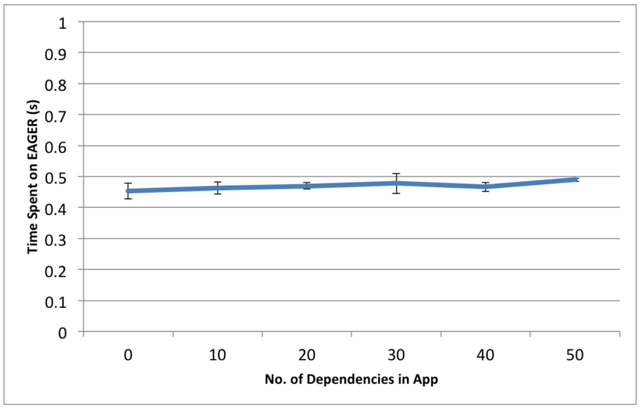

Figure 3.4: Average EAGER overhead vs. number of dependencies declared in the application. Each data point averages three executions, the error bars are two standard deviations, and the units are seconds.

In this case, the EAGER implementation processes all dependency-related information via batch operations. As a result, the number of web service calls and database queries that originate due to varying number of dependencies remains constant.

3.4.3 Impact of Number of Policies

So far we have conducted all our experiments without any active governance policies in the system. In this section, we report how EAGER overhead is influenced by the number of policies.

The overhead of policy validation is largely dependent on the actual policy content which is implemented as Python code. Since users may include any Python code (as long as it falls in the accepted subset) in a policy file, evaluating a given policy can take an arbitrary amount of time. Therefore, in this experiment, our goal is to evaluate the overhead incurred by simply having many policy files to execute. We keep the content of the policies small and trivial. We create a policy file that runs following assertions:

1. Application name must start with an upper case letter

2. Application must be owned by a specific user

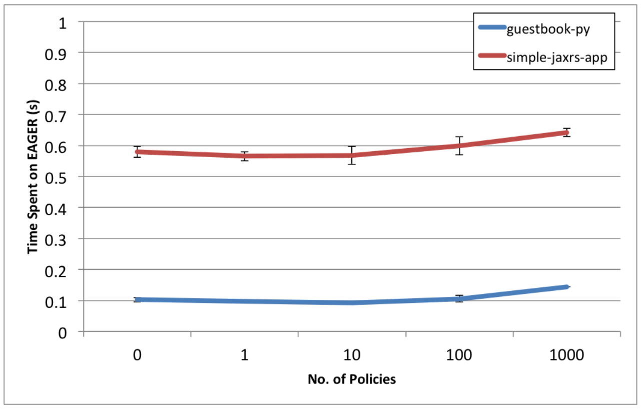

Figure 3.5: Average EAGER overhead vs. number of policies. Each data point averages three executions, the error bars are two standard deviations, and the units are seconds. Note that some of the error bars for guestbook-py are smaller than the graph features at this scale, and are thus obscured.

3. All API names must start with upper case letters

We create many copies of this initial policy file to vary the number of policies deployed. Then we evaluate the overhead of policy validation on two of our sample applications – guestbook-py and simple-jaxrs-app.

Figure 3.5 shows how the number of active policies impact EAGER overhead. We see that even large numbers of policies do not impact EAGER overhead significantly. It is only when the active policy count approaches 1000 that we can notice a small increase in the overhead. Even then, the increase in deployment time is under 0.1 seconds.

This result is due to the fact that EAGER loads policy content into memory at system startup, or when a new policy is deployed, and executes them from memory each time an application is deployed. Since policy files are typically small (at most a few kilobytes), this is a viable option. The overhead of validating the simple-jaxrs-app is higher than that of the guestbook-py because, simple-jaxrs-app exports web APIs. This means the third assertion in the policy set is executed for this app, and not for guestbook-py. Also, additional interactions with the metadata manager is needed in case of simple-jaxrs-app in order to persist the API metadata for future use.

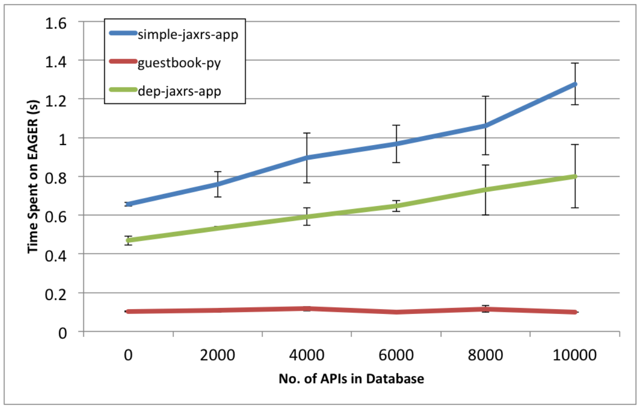

Figure 3.6: Average EAGER overhead vs. number of APIs in metadata manager. Each data point averages three executions, the error bars are two standard deviations, and the units are seconds. Note that some of the error bars for guestbook-py are smaller than the graph features at this scale and are thus obscured.

Our results indicate that EAGER scales well to hundreds of policies. That is, there is no significant overhead associated with simply having a large number of policy files. However, as mentioned earlier, the content of a policy may influence the overhead of policy validation, and will be specific to the policy and application EAGER analyzes.

3.4.4 Scalability

Next, we evaluate how EAGER scales when a large number of APIs are deployed in the cloud. In this experiment, we populate the EAGER metadata manager with a varying number of random APIs. We then attempt to deploy various sample applications. We also create random dependencies among the APIs recorded in the metadata manager to make the experimental setting more realistic.

Figure 3.6 shows that the deployment overhead of the guestbook-py application is not impacted by the growth of metadata in the cloud. Recall that guestbook-py does not export any APIs nor does it declare any dependencies. Therefore the deployment process of the guestbook-py application has minimal interactions with the metadata manager. Based on this result we conclude that applications that do not export web APIs are not significantly affected by the accumulation of metadata in EAGER.

Both simple-jaxrs-app and dep-jaxrs-app are affected by the volume of data stored in metadata manager. Since these applications export web APIs that must be recorded and validated by EAGER, the growth of metadata has an increasingly higher impact on them. The degradation of performance as a function of the number of APIs in the metadata manager database is due to the slowing of query performance of the RDBMS engine (MySQL) as the database size grows. Note that the simple-jaxrs-app is affected more by this performance drop, because it exports two APIs compared to the single API exported by dep-jaxrs-app. However, the growth in overhead is linear to the number of APIs deployed in the cloud, presumably indicating linear scaling factor in the installation of MySQL that EAGER used in these experiments. Also, even after deploying 10000 APIs, the overhead on simple-jaxrs-app is only increased by 0.5 seconds.

Another interesting characteristic in Figure 3.6 is the increase in overhead variance as the number of APIs in the cloud grows. We believe that this is due to the increasing variability of database query performance and the data transfer performance as the size of the database increases.

In summary, the current EAGER prototype scales well to 1000's of APIs. If further scalability is required, we can employ parallelization and database query optimization.

3.4.5 Experimental Results with a Real-World Dataset

Finally, we explore how EAGER operates with a real-world dataset with API metadata and dependency information. For this, we crawl the ProgrammableWeb API registry, and extract metadata regarding all registered APIs and mash-ups. At the time of the experiment, we managed to collect 11095 APIs and 7227 mash-ups, where each mash-up depends on one or more APIs.

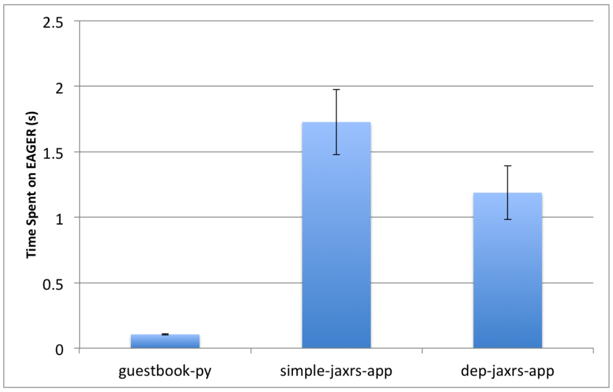

Figure 3.7: Average EAGER overhead over three experiments when deploying on ProgrammableWeb Dataset. The the error bars are two standard deviations, and the units are seconds.

We auto-generated API specifications for each API and mash-up, and populated the EAGER metadata manager with them. We then used the mashup-API dependency information gathered from ProgrammableWeb to register dependencies among the APIs in EAGER. This resulted in a dependency graph of total 18322 APIs with 33615 dependencies. We then deploy a subset of our applications, and measure EAGER overhead.

Figure 3.7 shows the results for three applications. The guestbook-py app (without any web APIs) is not significantly impacted by the large dependency database. Applications that export web APIs show a slightly higher deployment overhead due to the database scaling properties previously discussed. However, the highest overhead observed is under 2 seconds for simple-jaxrs-app, which is an acceptably small percentage of the 23.45 second deployment time as shown in table 3.2.

The applications in this experiment do not declare dependencies on any of the APIs in the ProgrammableWeb dataset. The dep-jaxrs-app does declare a dependency, but that is on an API exported by simple-jaxrs-app. To see how the deployment time is impacted when applications become dependent on other APIs already registered in EAGER, we deploy a test application that declares random fictitious dependencies on APIs from the ProgrammableWeb corpus registered in EAGER. We consider 10, 20, and 50

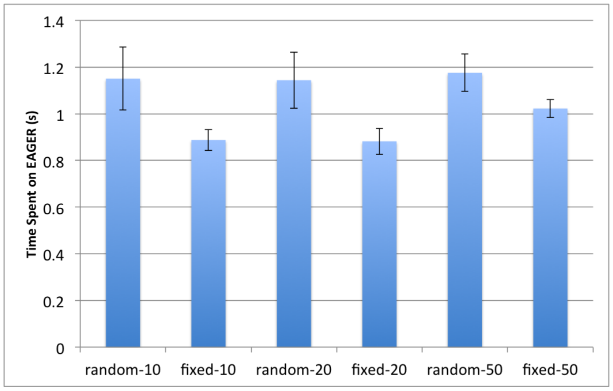

Figure 3.8: EAGER Overhead when deploying on ProgrammableWeb dataset with dependencies. The suffix value indicates the number of dependencies; the prefix indicates if these dependencies are randomized or not, upon redeployment. Each data point averages three executions, the error bars that are two standard deviations, and the units are seconds.

declared dependencies, and deploy each application three times. We present the results in Figure 3.8. For the "random" datasets, we run a deployment script that randomly modifies the declared dependencies at each redeployment. For the "fixed" datasets the declared dependencies remains the same across redeployments.

We observe that the dependency count does not have a significant impact on the overhead. The largest overhead observed is under 1.2 seconds for 50 randomly varied dependencies. In addition, when the dependency declaration is fixed, the overhead is slightly smaller. This is because our prototype caches the edges of its internally generated dependency tree, which expedites redeployments.

In summary, EAGER adds a very small overhead to the application deployment process, and this overhead increases linearly with the number of APIs exported by the applications, and the number of APIs deployed in the cloud. Interestingly, the number of deployed policies and declared dependencies have little impact on the EAGER governance overhead. Finally, our results indicate that EAGER scales well to 1000's of APIs and adds less than 2 seconds latency with over $18,000$ "real-world" deployed APIs in its database. Based on this analysis we conclude that enforced deployment-time API governance can be implemented in modern PaaS clouds with negligible overhead and high scalability. Further, deployment-time API governance can be made an intrinsic component of the PaaS cloud itself, thus alleviating the need for weakly integrated third-party API management solutions.

3.5 Related Work

Our research builds upon advances in the areas of SOA governance and service management. Guan et al introduced FASWSM [73] a web service management framework for application servers. FASWSM uses an adaptation technique that wraps web services in a way so they can be managed by the underlying application server platform. Wu et al introduced DART-Man [74], a web service management system based on semantic web concepts. Zhu and Wang proposed a model that uses Hadoop and HBase to store web service metadata, and process them to implement a variety of management functions [75]. Our work is different from these past approaches in that EAGER targets policy *enforcement*, and we focus on doing so by extending extant cloud platforms (e.g. PaaS) to provide an integrated and scalable governance solution.

Lin et al proposed a service management system for clouds that monitors all service interactions via special “hooks” that are connected to the cloud-hosted services [76]. These hooks monitor and record service invocations, and also provide an interface so that the individual service artifacts can be managed remotely. However, this system only supports run-time service management and provides no support for deployment-time policy checking and enforcement. Kikuchi and Aoki [77] proposed a technique based on model checking to evaluate the operational vulnerabilities and fault propagation patterns in cloud services. However, this system provides no active monitoring or enforcement functionality. Sun et al proposed a reference architecture for monitoring and managing cloud services [78]. This too lacks deployment-time governance, policy validation support, and the ability to intercept and act upon API calls which limits its use as a comprehensive governance solution for clouds.

Other researchers have shown that policies can be used to perform a wide range of governance tasks for SOA such as access control [79, 80], fault diagnosis [81], customization [82], composition [83, 84] and management [85, 86, 87]. We build upon the foundation of these past efforts, and use policies to govern RESTful web APIs deployed in cloud settings. Our work is also different in that it defines an executable policy language (implemented as a subset of Python in the EAGER prototype) that employs a simple, developer-friendly syntax based upon the Python language (vs XML), which is capable of capturing a wide range of governance requirements.

Peng, Lui and Chen showed that the major concerns associated with SOA governance involve retaining the high reliability of services, recording how many services are available on the platform to serve, and making sure all the available services are operating within an acceptable service level [20]. EAGER attempts to satisfy similar requirements for modern RESTful web APIs deployed in cloud environments. EAGER's metadata manager and ADP record and keep track of all deployed APIs in a simple, extensible, and comprehensive manner. Moreover, EAGER's policy validation, dependency management, and API change management features "fail fast" to detect violations immediately making diagnosis and remediation less complex, and prevent the system from ever entering a non-compliant state.

API management has been a popular topic in the industry over the last few years, resulting in many commercial and open source API management solutions [72, 46, 47, 88]. These products facilitate API lifecycle management, traffic shaping, access control, monitoring and a variety of other important API-related functionality. However, these tools do not support deep integration with cloud environments in which many web applications and APIs are deployed today. EAGER is also different in that it combines deployment-time and run-time enforcement. Previous systems either work exclusively at run-time or do not include an enforcement capability (i.e. they are advisory).

3.6 Conclusions and Future Work

In this chapter, we describe EAGER, a model and a software architecture that facilitates API governance as a cloud-native feature. EAGER supports comprehensive policy enforcement, dependency management, and a variety of other deployment-time API governance features. It promotes many software development and maintenance best practices including versioning, code reuse, and API backwards compatibility retention. EAGER also includes a language based on Python that enables creating, debugging, and maintaining API governance policies in a simple and intuitive manner. EAGER can be built into cloud platforms that are used to host APIs to automate governance tasks that otherwise require custom code or developer intervention.

Our empirical results, gathered using a prototype of EAGER developed for AppScale, show that EAGER adds negligibly small overhead to the cloud application deployment process, and the overhead grows linearly with the number of APIs deployed. We also show that EAGER scales well to handle tens of thousands of APIs and hundreds of policies. Based on our results we conclude that efficient and automated policy enforcement is feasible in cloud environments. Furthermore, we find that policy enforcement at deployment-time can help cloud administrators and application developers achieve administrative conformance and developer best practices with respect to cloud-hosted web applications.

As part of our future work, we plan to investigate the degree to which deployment-time governance can be expanded. Run-time API governance imposes a number of new scalability and reliability challenges. By offloading as much of the governance overhead to deployment-time as possible, EAGER ensures that the impact of run-time governance is minimized.

We also plan to investigate the specific language features that are essential to EAGER's combined deployment-time and run-time approach. The use of Python in the prototype proved convenient from a programmer productivity perspective. It is not yet clear, however, whether the full set of language features that we have left unrestricted are necessary. By minimizing the policy language specification we hope to make its implementation more efficient, less error prone to develop and debug, and more amenable to automatic analysis.

Another future research direction is the integration of policy language and run-time API governance. We wish to explore the possibility of using the same Python-based policy language for specifying policies that are enforced on APIs at run-time (i.e. on individual API calls). Since API calls far more frequent than API deployment events, we should evaluate the performance aspects of the policy engine to make this integration practically useful.

Chapter 4

Response Time Service Level Objectives for Cloud-hosted Web Applications

In the previous chapter we discussed how to implement API governance in cloud environments via policy enforcement. This chapter focuses on stipulating bounds on the performance of cloud-hosted web applications. The ability to understand the performance bounds of an application is vital in several governance use cases such as performance-aware policy enforcement, and application performance monitoring.

Cloud-hosted web applications are deployed and used as web services. They enable a level of service reuse that both expedites and simplifies the development of new client applications. Despite the many benefits, reusing existing services also has pitfalls. In particular, new client applications become dependent on the services they compose. These dependencies impact correctness, performance, and availability of the composite applications, for which the "top level" developer is often held accountable. Compounding the situation, the underlying services can and do change over time while their APIs remain stable, unbeknownst to the developers that programmatically access them. Unfortunately, there is a dearth of tools that help developers reason about these dependencies throughout an application's life cycle (i.e. development, deployment, and run-time). Without such tools, programmers must adopt extensive, continuous, and costly, testing and profiling methods to understand the performance impact on their applications that results from the increasingly complex collection of services that they depend on.

We present Cerebro to address this requirement without subjecting applications to extensive testing or instrumentation. Cerebro is a new approach that predicts bounds on the response time performance of web APIs exported by applications that are hosted in a PaaS cloud. The goal of Cerebro is to allow a PaaS administrator to determine what response time service level objective (SLO) can be fulfilled by each web API operation exported by the applications hosted in the PaaS.

An "SLO" specifies the minimum service level promised by the service provider regarding some non-functional property of the service such as its availability or performance (response time). Such SLOs are explicitly stated by the service provider, and are typically associated with a correctness probability, which can be described as the likelihood the service will meet the promised minimum service level. A typical availability SLO takes the form: "the service will be available $p\%$ of the time". Here the value $p\%$ is the correctness probability of the SLO. Similarly, a response time SLO would take the form of the statement: "the service will respond under Q milliseconds, $p\%$ of the time. Naturally, p should be a value close to 100, for this type of SLOs to be useful in practice.

In a corporate setting, SLOs are used to form service level agreements (SLAs), formal contracts that govern the service provider-consumer relationship [89]. They consist of SLOs, and the clauses that describe what happens if the service fails to meet those SLOs (for example, if the service is only available $p'\%$ of the time, where $p' < p$). This typically boils down to service provider paying some penalty (a refund), or providing some form of free service credits for the users. We do not consider such legal and social obligations of an SLA in this work, and simply focus on the minimum service levels (i.e. SLOs), and the associated correctness probabilities, since those are the parameters that matter from a performance and capacity planning point of view of an application.

Currently, cloud computing systems such as Amazon Web Services (AWS) [3] and Google App Engine (GAE) [4] advertise SLOs specifying the fraction of availability over a fixed time period (i.e. uptime) for their services. However, they do not provide SLOs that state minimum levels of performance. In contrast, Cerebro facilitates auto-generating performance SLOs for cloud-hosted web APIs in a way that is scalable. Cerebro uses a combination of static analysis of the hosted web APIs, and runtime monitoring of the PaaS kernel services to determine what minimum statistical guarantee can be made regarding an API's response time, with a target probability specified by a PaaS administrator. These calculated SLOs enable developers to reason about the performance of the client applications that consume the cloud-hosted web APIs. They can also be used to negotiate SLAs concerning the performance of cloud-hosted web applications. Moreover, predicted SLOs are useful as baselines or thresholds when monitoring APIs for consistent performance – a feature that is useful for both API providers and consumers. Collectively, Cerebro and the SLOs predicted by it enable implementing a number of automated governance scenarios involving policy enforcement and application performance monitoring, in ways that were not possible before.

Statically reasoning about the execution time of arbitrary programs is challenging if not unsolvable. Therefore we scale the problem down by restricting our analysis to cloud-hosted web applications. Specifically, Cerebro generates response time SLOs for APIs exported by a web application developed using the kernel services available within a PaaS cloud. For brevity, in this work we will use the term *web API* to refer to a web-accessible API exported by an application hosted on a PaaS platform. Further, we will use the term *kernel services* to refer to the services that are maintained as part of the PaaS and available to all hosted applications. This terminology enables us to differentiate the internal services of the PaaS from the APIs exported by the deployed applications. For example, an application hosted in Google App Engine might export one or more web APIs to its users while leveraging the internal datastore kernel service that is available as part of the Google App Engine PaaS.

Cerebro uses static analysis to identify the PaaS kernel invocations that dominate the response time of web APIs. By surveying a collection of web applications developed for a PaaS cloud, we show that such applications indeed spend majority of their execution time on PaaS kernel invocations. Further, they do not have many branches and loops, which makes them amenable to static analysis (section 4.1). Independently, Cerebro also maintains a running history of response time performance for PaaS kernel services. It uses QBETS [90] – a forecasting methodology we have developed in prior work for predicting bounds on "ill behaved" univariate time series – to predict response time bounds on each PaaS kernel invocation made by the application. It combines these predictions dynamically for each static program path through a web API operation, and returns the "worst-case" upper bound on the time necessary to complete the operation.

Because service implementations and platform behavior under load change over time, Cerebro's predictions necessarily have a lifetime. That is, the predicted SLOs may become invalid after some time. As part of this chapter, we develop a model for detecting such SLO invalidations. We use this model to investigate the effective lifetime of Cerebro predictions. When such changes occur, Cerebro can be reinvoked to establish new SLOs for any deployed web API.

We have implemented Cerebro for both the Google App Engine public PaaS, and the AppScale private PaaS. Given its modular design and this experience, we believe that Cerebro can be easily integrated into any PaaS system. We use our prototype implementation to evaluate the accuracy of Cerebro, as well as the tightness of the bounds it predicts (i.e. the difference between the predictions and the actual API execution times). To this end, we carry out a range of experiments using App Engine applications that are available as open source.

We also detail the duration over which Cerebro predictions hold in both GAE and AppScale. We find that Cerebro generates correct SLOs (predictions that meet or exceed their probabilistic guarantees), and that these SLOs are valid over time periods ranging from 1.4 hours to several weeks. We also find that the high variability of performance in public PaaS clouds due to their multi-tenancy and massive scale requires that Cerebro be more conservative in its predictions to achieve the desired level of correctness. In comparison, Cerebro is able to make much tighter SLO predictions for web APIs hosted in private, single tenant clouds.

Because Cerebro provides this analysis statically it imposes no run-time overhead on the applications themselves. It requires no run-time instrumentation of application code, and it does not require any performance testing of the web APIs. Furthermore, because the PaaS is scalable and platform monitoring data is shared across all Cerebro executions, the continuous monitoring of the kernel services generates no discernible load on the cloud platform. Thus we believe Cerebro is suitable for highly scalable cloud settings.

Finally, we have developed Cerebro for use with EAGER (**E**nforced **A**PI **G**overnance **E**ngine for **R**EST) [91] – an API governance system for PaaS clouds. EAGER attempts to enforce governance policies at the deployment-time of cloud applications. These governance policies are specified by cloud administrators to ensure the reliable operation of the cloud and the deployed applications. PaaS platforms include an application deployment phase during which the platform provisions resources for the application, installs the application components, and configures them to use the kernel services. EAGER injects a policy checking and enforcement step into this deployment workflow, so that only applications that are compliant with respect to site-specific policies are successfully deployed. Cerebro allows PaaS administrators to define performance-aware EAGER policies that allow an application to be deployed *only* when its web APIs meet a pre-determined SLO, and developers to be notified by the platform when such SLOs require revision.

We structure the rest of this chapter as follows. We first characterize the domain of PaaS-hosted web APIs for GAE and AppScale in Section 4.1. We then present the design of Cerebro in section 4.2 and overview our software architecture and prototype implementation. Next, we present our empirical evaluation of Cerebro in section 4.3. Finally, we discuss related work (Section 4.4) and conclude (Section 4.5).

4.1 Domain Characteristics and Assumptions

The goal of our work is to analyze a web API statically, and from this analysis without deploying or running the web API, accurately predict an upper bound on its response time. With such a prediction, developers and cloud administrators can provide performance SLOs to the API consumers (human or programmatic), to help them reason about the performance implications of using APIs – something that is not possible today. For general purpose applications, such worst-case execution time analysis has been shown by numerous researchers to be challenging to achieve for all but simple programs or specific application domains. To overcome these challenges, we take inspiration from the latter, and exploit the application domain of PaaS-hosted web APIs to achieve our goal. In this chapter, we focus on the popular Google App Engine (GAE) public PaaS, and AppScale private PaaS, which support the same applications, development and deployment model, and platform services.

The first characteristic of PaaS systems that we exploit to facilitate our analysis is their predefined programming interfaces through which they export various kernel services. Herein we refer to these programming interfaces as the cloud software development kit or the *cloud SDK*. The cloud SDK is comprised of several interfaces, each of which plays the role of a client stub for some kernel service offered by the cloud platform. We refer to the individual member interfaces of the cloud SDK as *cloud SDK interfaces*, and to their constituent operations as *cloud SDK operations*. These interfaces export scalable functionality that is commonly used to implement web APIs: key-value datastores, caching, task scheduling, security and authentication, etc. In an application, each cloud SDK call represents an invocation of a PaaS kernel service. Therefore, we use the terms *cloud SDK calls* and *PaaS kernel invocations* interchangeably in the remainder of this chapter. The App Engine and AppScale cloud SDK is detailed in `https://cloud.google.com/appengine/docs/java/javadoc/`.

With PaaS clouds, developers implement their application code as a combination of calls to the cloud SDK, and their own code. Developers then upload their applications to the cloud for deployment. Once deployed, the applications and any web APIs exported by them can be accessed via HTTP/S requests by external or co-located clients.

Typically, PaaS-hosted web APIs make one or more cloud SDK calls. The reason for this is two-fold. First, kernel services that underpin the cloud SDK provide web APIs with much of the functionality that they require. Second, PaaS clouds “sandbox” web APIs to enforce quotas, to enable billing, and to restrict certain functionality that can lead to security holes, platform instability, or scaling issues [92]. For example, GAE and AppScale cloud platforms restrict the application code from accessing the local file system, accessing shared memory, using certain libraries, and arbitrarily spawning threads. Therefore developers must use the provided cloud SDK operations to implement program logic equivalent to the restricted features. For example, the datastore interface can be used to read and write persistent data instead of using the local file system, and the memcache interface can be used in lieu of global shared memory.

Furthermore, the only way for a web API to execute is in response to an HTTP/S request or as a background task. Therefore, execution of all web API operations start and end at well defined program points, and we are able to infer this structure from common software patterns. Also, concurrency is restricted by capping the number of threads and requiring that a thread cannot outlive the request that creates it. Finally, PaaS clouds enforce quotas and limits on kernel service (cloud SDK) use [93, 94, 92]. App Engine, for example, requires that all web API requests complete under 60 seconds. Otherwise they are terminated by the platform. Such enforcement places a strict upper bound on the execution of a web API operation.

To understand the specific characteristics of PaaS-hosted web APIs, and the potential of this restricted domain to facilitate efficient static analysis and response time prediction, we next summarize results from static analysis (using the Soot framework [95]) of 35 real world App Engine web APIs. These web APIs are open source (available via GitHub [96]), written in Java, and run over Google App Engine or AppScale without modification. We selected them based on availability of documentation, and the ability to compile and run them without errors.

Our analysis detected a total of 1458 Java methods in the analyzed codes. Figure 4.1 shows the cumulative distribution of static program paths in these methods. Approximately 97% of the methods considered in the analysis have 10 or fewer static program paths through them. 99% of the methods have 36 or fewer paths. However, the CDF is heavy tailed, and grows to 34992. We truncate the graph at 100 paths for clarity. As such, only a very small number of methods each contains a large number of paths. Fortunately, over 65% of the methods have exactly 1 path (i.e. there are no branches).

Next, we consider the looping behavior of web APIs. 1286 of the methods (88%) considered in the study do not have any loops. 172 methods (12%) contain loops. We

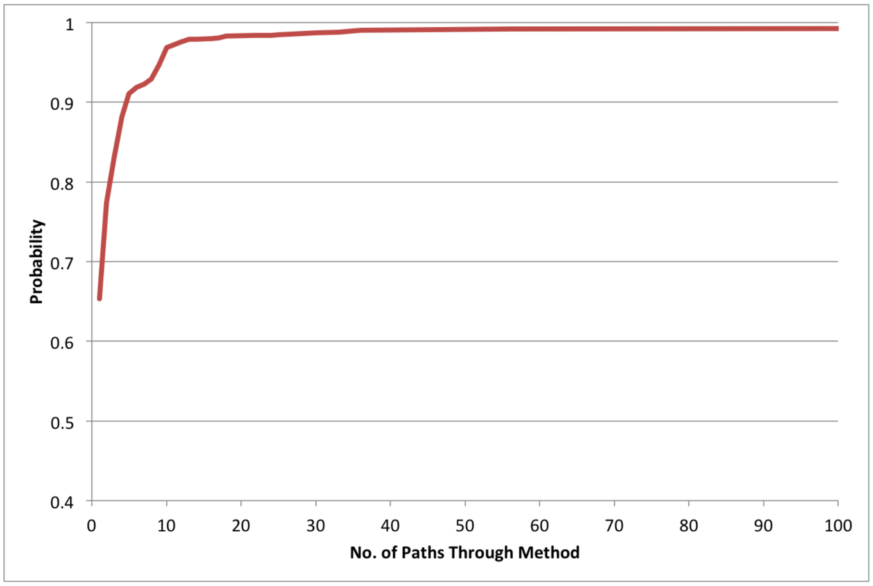

Figure 4.1: CDF of the number of static paths through methods in the surveyed web APIs.

believe that this characteristic is due to the fact that the PaaS SDK and the platform restrictions like quotas and response time limits discourage looping.

Approximately 29% of all the loops in the analyzed programs do not contain any cloud SDK calls. A majority of the loops (61%) however, are used to iterate over a dataset that is returned from the datastore cloud SDK interface of App Engine (i.e iterating on the result set returned by a datastore query). We refer to this particular type of loops as *iterative datastore reads*.

Table 4.1 lists the number of times each cloud SDK interface is called across all paths and methods in the analyzed programs. The datastore API is the most commonly used interface. This is because data management is fundamental to most web APIs, and the PaaS disallows using the local filesystem to do so for scalability and portability reasons.

Next, we explore the number of cloud SDK calls made along different paths of execution in the web APIs. For this study we consider all paths of execution through the methods (64780 total paths). Figure 4.2 shows the cumulative distribution of the number of SDK calls within paths. Approximately 98% of the paths have 1 cloud SDK call or

Table 4.1: Static cloud SDK calls in surveyed web APIs

Cloud SDK Interface	No. of Invocations
blobstore	7
channel	1
datastore	735
files	4
images	3
memcache	12
search	6
taskqueue	24
tools	2
urlfetch	8
users	44
xmpp	3

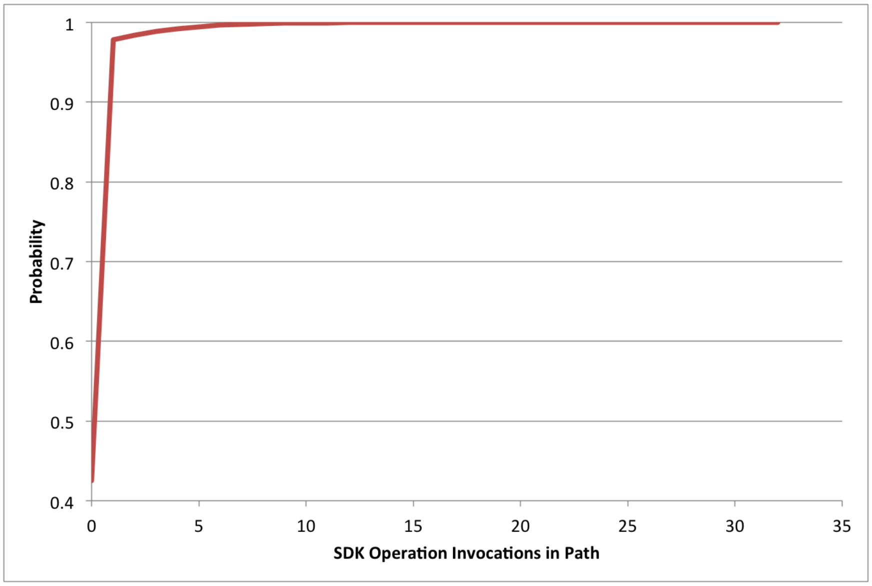

Figure 4.2: CDF of cloud SDK call counts in paths of execution.

fewer. The probability of finding an execution path with more than 5 cloud SDK calls is smaller than 1%.

Finally, our experience with App Engine web APIs indicates that a significant portion of the total time of a method (web API operation) is spent in cloud SDK calls. Confirming this hypothesis requires careful instrumentation (i.e. difficult to automate) of the web API codes. We performed such a test by hand on two representative applications, and found that the time spent in code other than cloud SDK calls accounts for 0-6% of the total time (0-3ms for a 30-50ms web API operation).

This study of various characteristics typical of PaaS-hosted web APIs indicates that there may be opportunities to exploit the specific aspects of this application domain to simplify analysis, and to facilitate performance SLO prediction. In particular, operations in these applications are short, have a small number of paths to analyze, implement few loops, and invoke a small number of cloud SDK calls. Moreover, most of the time spent executing these operations results from cloud SDK invocations. In the next section, we describe our design and implementation of Cerebro that takes advantage of these characteristics and assumptions. We then use a Cerebro prototype to experimentally evaluate its efficacy for estimating the worst-case response time for applications from this domain.

4.2 Cerebro

Given the restricted application domain of PaaS-hosted web APIs, we believe that it is possible to design a system that predicts response time SLOs for them using only static information from the web API code itself. To enable this, we design Cerebro with three primary components:

- A static analysis tool that extracts sequences of cloud SDK calls for each path through a method (web API operation),

- A monitoring agent that runs in the target PaaS, and efficiently monitors the performance of the underlying cloud SDK operations, and
- An SLO predictor that uses the outputs of these two components to accurately predict an upper bound on the response time of the web API.

We overview each of these components in the subsections that follow, and then discuss the Cerebro workflow with an example.

4.2.1 Static Analysis

This component analyzes the source code of the web API (or some intermediate representation of it), and extracts a sequence of cloud SDK calls. We implement our analysis for Java bytecode programs using the Soot framework [95]. Currently, our prototype analyzer considers the following Java codes as exposed web APIs.

- classes that extend the *javax.servlet.HttpServlet* class (i.e. Java servlet implementations)
- classes that contain JAX-RS *@Path* annotations, and
- any other classes explicitly specified by the developer in a special configuration file.

Cerebro performs a simple construction and inter-procedural static analysis of control flow graph (CFG) [97, 98, 99, 100] for each web API operation. The algorithm extracts all cloud SDK calls along each path through the methods. Cerebro analyzes other functions that the method calls, recursively. Cerebro caches cloud SDK details for each function once analyzed so that it can be reused efficiently for other call sites to the same function. Cerebro does not analyze third-party library calls, if any, which in our experience typically do not contain cloud SDK calls. Cerebro encodes each cloud SDK call sequence for each path in a lookup table. We identify cloud SDK calls by their Java package name (e.g. `com.google.appengine.apis`).

To handle loops, we first extract them from the CFG and annotate all cloud SDK calls that occur within them. We then annotate each such SDK call with an estimate on the number of times the loop is likely to execute in the worst case. We estimate loop bounds using a loop bound prediction algorithm based on abstract interpretation [101].

As shown in the previous section, loops in these programs are rare, and when they do occur, they are used to iterate over a dataset returned from a database. For such data-dependent loops, we estimate the bounds if specified in the cloud SDK call (e.g. the maximum number of entities to return [102]). If our analysis is unable to estimate the bounds for these loops, Cerebro prompts the developer for an estimate of the likely dataset size and/or loop bounds.

4.2.2 PaaS Monitoring Agent

Cerebro monitors and records the response time of individual cloud SDK operations within a running PaaS system. Such support can be implemented as a PaaS-native feature or as a PaaS application (web API); we use the latter in our prototype. The monitoring agent runs in the background with, but separate from, other PaaS-hosted web APIs. The agent invokes cloud SDK operations periodically on synthetic datasets, and records timestamped response times in the PaaS datastore for each cloud SDK operation. The agent also periodically reclaims old measurement data to eliminate unnecessary storage. The Cerebro monitoring and reclamation rates are configurable, and monitoring benchmarks can be added and customized easily to capture common PaaS-hosted web API coding patterns.

In our prototype, the agent monitors the datastore and memcache SDK interfaces every 60 seconds. In addition, it benchmarks loop iteration over datastore entities to capture the performance of iterative datastore reads for datastore result set sizes of 10, 100, and 1000. We limit ourselves to these values because the PaaS requires that all operations complete (respond) within 60 seconds – so the data sizes (i.e. number of data entities) returned are typically small. Sizing up the datastore in terms of powers of 10, mirrors the typical approach taken by DevOps personnel to approximate the size of a database. If necessary, our prototype allows adding iterative datastore read benchmarks for other result set sizes easily.

4.2.3 Making SLO Predictions

To make SLO predictions, Cerebro uses Queue Bounds Estimation from Time Series (QBETS) [90], a non-parametric time series analysis method that we developed in prior work. We originally designed QBETS for predicting the scheduling delays for the batch queue systems used in high performance computing environments, but it has proved effective in other settings where forecasts from arbitrary times series are needed [103, 104, 105]. In particular, it is both non-parametric, and it automatically adapts to changes in the underlying time series dynamics making it useful in settings where forecasts are required from arbitrary data with widely varying characteristics. We adapt it herein for use "as-a-service" in PaaS systems to predict the execution time of web APIs.

A QBETS analysis requires three inputs:

1. A time series of data generated by a continuous experiment.

2. The percentile for which an upper bound should be predicted ($p \in [1..99]$).

3. The upper confidence level of the prediction ($c \in (0, 1)$).

QBETS uses this information to predict an upper bound for the p^{th} percentile of the time series. It does so by treating each observation in the time series as a Bernoulli trial with probability $0.01p$ of success. Let $q = 0.01p$. If there are n observations, the probability of there being exactly k successes is described by a Binomial distribution (assuming observation independence) having parameters n and q. If Q is the p^{th} percentile of the distribution from which the observations have been drawn, the equation

$$1 - \sum_{j=0}^{k} \binom{n}{j} \cdot (1-q)^j \cdot q^{n-j} \tag{4.1}$$

gives the probability that more than k observations are greater than Q. As a result, the k^{th} largest value in a sorted list of n observations gives an upper c confidence bound on Q when k is the smallest integer value for which Equation 4.1 is larger than c.

More succinctly, QBETS sorts observations in a history of observations, and computes the value of k that constitutes an index into this sorted list that is the upper c confidence bound on the p^{th} percentile. The methodology assumes that the time series of observations is ergodic so that, in the long run, the confidence bounds are accurate.

QBETS also attempts to detect change points in the time series of observations so that it can apply this inference technique to only the most recent segment of the series that appears to be stationary. To do so, it compares percentile bound predictions with observations throughout the series, and determines where the series is likely to have undergone a change. It then discards observations from the series prior to this change point and continues. As a result, when QBETS starts, it must "learn" the series by scanning it in time series order to determine the change points. We report Cerebro learning time in our empirical evaluation in subsection 4.3.6.

Note that c is an upper confidence level on p^{th} percentile which makes the QBETS bound estimates conservative. That is, the value returned by QBETS as a bound prediction is larger than the true p^{th} percentile with probability $1 - c$ under the assumptions of the QBETS model. In this study, we use the 95^{th} percentile with $c = 0.01$.

Note that the algorithm itself can be implemented efficiently so that it is suitable for on-line use. Details of this implementation as well as a fuller accounting of the statistical properties and assumptions are available in [90, 103, 104, 106].

QBETS requires a sufficiently large number of data points in the input time series before it can make an accurate prediction. Specifically, the largest value in a sorted list of n observations is greater than the p^{th} percentile with confidence c when $n >= log(c)/log(0.01p)$.

For example, predicting the 95^{th} percentile of the API execution time, with an upper confidence of 0.01 requires at least 90 observations. We use this limit as a lower bound for the length of the history to keep. There is no upper bound for the history length that QBETS can process. But in Cerebro's case, several thousand data points in the history (i.e. 1-3 days of monitoring data) provides a good balance between results accuracy and computation overhead.

The minimum history length also provides a bound on the variability of the time series that can be tolerated by QBETS. In general, each time series must be approximately ergodic, meaning their mean and the variance should not change abruptly. More specifically, if the values in the time series change too fast for QBETS to gather a stationary dataset at least as long as the minimum history length, its prediction accuracy may suffer.

4.2.4 Example Cerebro Workflow

Figure 4.3 illustrates how the Cerebro components interact with each other during the prediction making process. Cerebro can be invoked when a web API is deployed to

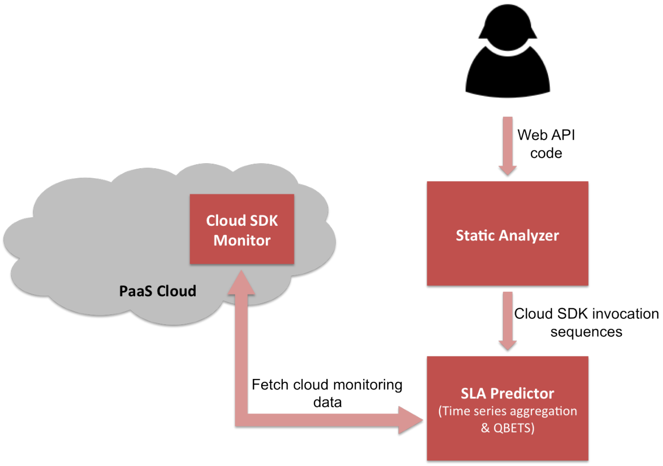
Web API code
Static Analyzer
Cloud SDK invocation sequences
SLA Predictor
(Time series aggregation & QBETS)
Cloud SDK Monitor
PaaS Cloud
Fetch cloud monitoring data

Figure 4.3: Cerebro architecture and component interactions.

a PaaS cloud, or at any time during the development process to give developers insight into the worst-case response time of their applications.

Upon invoking Cerebro with a web API code, Cerebro performs its static analysis on all operations in the API. For each analyzed operation it produces a list of annotated cloud SDK invocation sequences – one sequence per program path. Cerebro then prunes this list to eliminate duplicates. Duplicates occur when a web API operation has multiple program paths with the same sequence of cloud SDK invocations. Next, for each pruned list Cerebro performs the following operations:

1. Retrieve (possibly compressed) benchmarking data from the monitoring agent for all SDK operations in each sequence. The agent returns ordered time series data (one time series per cloud SDK operation).

2. Align retrieved time series across operations in time, and sum the aligned values to form a single *joint* time series of the summed values for the sequence of cloud SDK operations.

3. Run QBETS on the joint time series with the desired p and c values to predict an

upper bound.

Cerebro uses the largest predicted value (across path sequences) as its SLO prediction for a web API operation. The exhaustive approach by which Cerebro predicts SLOs for all possible program paths ensures that the final SLO holds valid regardless of which path gets executed at runtime. This SLO prediction process can be implemented as a co-located service in the PaaS cloud or as a standalone utility. We do the latter in our prototype.

As an example, suppose that the static analysis results in the cloud SDK invocation sequence $< op_1, op_2, op_3 >$ for some operation in a web API. Assume that the monitoring agent has collected the following time series for the three SDK operations:

- op_1: $[t_0 : 5, t_1 : 4, t_2 : 6, ..., t_n : 5]$
- op_2: $[t_0 : 22, t_1 : 20, t_2 : 21, ..., t_n : 21]$
- op_3: $[t_0 : 7, t_1 : 7, t_2 : 8, ..., t_n : 7]$

Here t_m is the time at which the m^{th} measurement is taken. Cerebro aligns the three time series according to timestamps, and sums the values to obtain the following joint time series: $[t_0 : 34, t_1 : 31, t_2 : 35, ..., t_n : 33]$

If any operation is tagged as being inside a loop, where the loop bounds have been estimated, Cerebro multiplies the time series data corresponding to that operation by the loop bound estimate before aggregating. In cases where the operation is inside a data-dependent loop, we request the time series data from the monitoring agent for its iterative datastore read benchmark for a number of entities that is equal to or larger than the annotation, and include it in the joint time series.

Cerebro passes the final joint time series for each sequence of operations to QBETS, which returns the worst-case upper bound response time it predicts. If the QBETS predicted value is Q milliseconds, Cerebro forms the SLO as "the web API will respond in under Q milliseconds, $p\%$ of the time". When the web API has multiple operations, Cerebro estimates multiple SLOs for the API. If a single value is needed for the entire API regardless of operation, Cerebro returns the largest predicted value as the final SLO (i.e. the worst-case SLO for the API).

4.2.5 SLO Durability

For a given web API, Cerebro predicts an initial response time SLO at the API's deployment-time (following the above workflow). It then consults an on-line API benchmarking service to continuously verify the predicted response time SLO to determine if and when it has been violated. SLO violations occur when conditions in the PaaS change in ways that adversely impact the performance of the cloud SDK operations. Such changes can result from congestion (multi-tenancy), component failures, and modifications to PaaS service implementations. The continuous tracking of SLO violations is necessary to notify the affected API consumers promptly.

Cerebro also periodically recomputes the SLOs for the APIs over time. Cerebro is able to perform fast, online prediction of time series percentiles via QBETS as more SDK benchmarking data becomes available from the cloud SDK monitor. This periodic recomputation of SLOs is important because changes in the PaaS can occur that make new SLOs available that are better and tighter than the previously predicted ones. Cerebro must detect when such changes occur so that API consumers can be notified.

To determine SLO durability, we extend Cerebro with a statistical model for detecting when a Cerebro-generated SLO becomes invalid. Suppose at time t Cerebro predicts value Q as the p-th percentile of some API's execution time. If Q is a correct prediction, the probability of API's next measured response time being greater than Q is $1 - (0.01p)$. If the time series consists of independent measurements, then the probability of seeing n consecutive values greater than Q (due to random chance) is $(1 - 0.01p)^n$. For example, using the 95^{th} percentile, the probability of seeing 3 values in a row larger than the predicted percentile due to random chance is $(0.05)^3 = 0.00012$.

This calculation is conservative with respect to autocorrelation. That is, if the time series is stationary but autocorrelated, then the number of consecutive values above the 95^{th} percentile that correspond to a probability of 0.00012 is larger than 3. For example, in previous work [90] using an artificially generated AR(1) series, we observed that 5 consecutive values above the 95^{th} percentile occurred with probability 0.00012 when the first autocorrelation was 0.5, and 14 when the first autocorrelation was 0.85. QBETS uses a look-up table of these values to determine the number of consecutive measurements above Q that constitute a "rare event" indicating a possible change in conditions.

Each time Cerebro makes a new prediction, it computes the current autocorrelation, and uses the QBETS rare-event look-up table to determine C_w: the number of consecutive values that constitute a rare event. We measure the time from when Cerebro makes the prediction until we observe C_w consecutive values above that prediction as being the time duration over which the prediction is valid. We refer to this duration as the *SLO validity duration*.

4.2.6 SLO Reassessment

We extend Cerebro with an SLO reassessment process that invalidates SLOs at the end of the SLO validity duration, and provides a new SLO for the API consumer. API consumers receive an initial SLO for a web API hosted by a Cerebro-equipped PaaS as part of the API subscription process (i.e. when obtaining API keys). This initial SLO may be issued in the form of an SLA that is negotiated between the API provider and the consumer. At this point Cerebro records the tuple $< Consumer, API, Timestamp, SLO >$.

When Cerebro detects consecutive violations of one of its predictions, it considers the corresponding SLO to be invalid, and provides the affected API consumers with a new SLO. Upon this SLO change, Cerebro updates the $Timestamp$ and SLO entries in the appropriate data tuple for future reference.

There is also a second way that an API consumer may encounter an SLO change. When recomputing SLOs periodically, Cerebro might come across situations where the latest SLO is smaller than some previously issued SLO (i.e. a tighter SLO is available). Cerebro can notify the API consumer about this prospect. If the API consumer consents to the SLO change, Cerebro may update the data tuple, and treat the new SLO as in effect.

We next use empirical testing and simulations to explore the feasibility of the Cerebro SLO reassessment process, and evaluate how SLO validity duration and invalidation impact API consumers over time.

4.3 Experimental Results

To empirically evaluate Cerebro, we conduct experiments using five open source, Google App Engine applications.

StudentInfo RESTful (JAX-RS) application for managing students of a class (adding, removing, and listing student information).

ServerHealth Monitors, computes, and reports statistics for server uptime for a given web URL.

SocialMapper A simple social networking application with APIs for adding users and comments.

StockTrader A stock trading application that provides APIs for adding users, registering companies, buying and selling stocks among users.

Rooms A hotel booking application with APIs for registering hotels and querying available rooms.

These web APIs use the datastore cloud SDK interface extensively. The Rooms web API also uses the memcache interface. We focus on these two interfaces exclusively in this study. We execute these applications in the Google App Engine public cloud (SDK v1.9.17) and in an AppScale (v2.0) private cloud. We instrument the programs to collect execution time statistics for verification purposes only (the instrumentation data is not used to predict the SLOs). The AppScale private cloud used for testing was hosted using four "m3.2xlarge" virtual machines running on a private Eucalyptus [6] cloud.

We first report the time required for Cerebro to perform its analysis and SLO prediction. Across web APIs, Cerebro takes 10.00 seconds on average, with a maximum time of 14.25 seconds for the StudentInfo application. These times include the time taken by the static analyzer to analyze all the web API operations, and the time taken by QBETS to make predictions. For these results, the length of the time series collected by PaaS monitoring agent is 1528 data points (25.5 hours of monitoring data). Since the QBETS analysis time depends on the length of the input time series, we also measured the time for 2 weeks of monitoring data (19322 data points) to provide some insight into the overhead of SLO prediction. Even in this case, Cerebro requires only 574.05 seconds (9.6 minutes) on average.

4.3.1 Correctness of Predictions

We first evaluate the correctness of Cerebro predictions. A set of predictions is *correct* if the *fraction* of measured response time values that fall below the Cerebro prediction is greater than or equal to the SLO success probability. For example, if the SLO success probability is 0.95 (i.e. $p = 95$ in QBETS) for a specific web API, then the Cerebro predictions are correct if at least 95% of the response times measured for the web API are smaller than their corresponding Cerebro predictions.

We benchmark each web API for a period of 15 to 20 hours. During this time we run a remote HTTP client that makes requests to the web APIs once every minute. The application instrumentation measures and records the response time of the API operation for each request (i.e. within the application). Concurrently, and within the same PaaS system, we execute the Cerebro PaaS monitoring agent, which is an independently hosted application within the cloud that benchmarks each SDK operation once every minute.

Our test request rate (1 request/minute) is not sufficient to put the backend cloud servers under any stress. However, cloud platforms like Google App Engine and AppScale are highly scalable. When the load increases, they automatically spin up new backend servers, and maintain the average response time of deployed web APIs steady. This enables us to measure and evaluate the correctness of the Cerebro predictions under light load conditions. Note that our cloud SDK benchmarking rate at the cloud SDK monitor is also 1 request per minute. We assume that the time series of cloud SDK performance is ergodic (i.e. stationary over a long period). Under that assumption, QBETS is insensitive to the measurement frequency, and a higher benchmarking rate would not significantly change the results.

Cerebro predicts the web API execution times using only the cloud SDK benchmarking data collected by Cerebro's PaaS monitoring agent. We configure Cerebro to predict an upper bound for the 95^{th} percentile of the web API response time, with an upper confidence of 0.01.

QBETS generates a prediction for *every* value in the input time series (one per minute). Cerebro reports the last one as the SLO prediction to the user or PaaS admin-

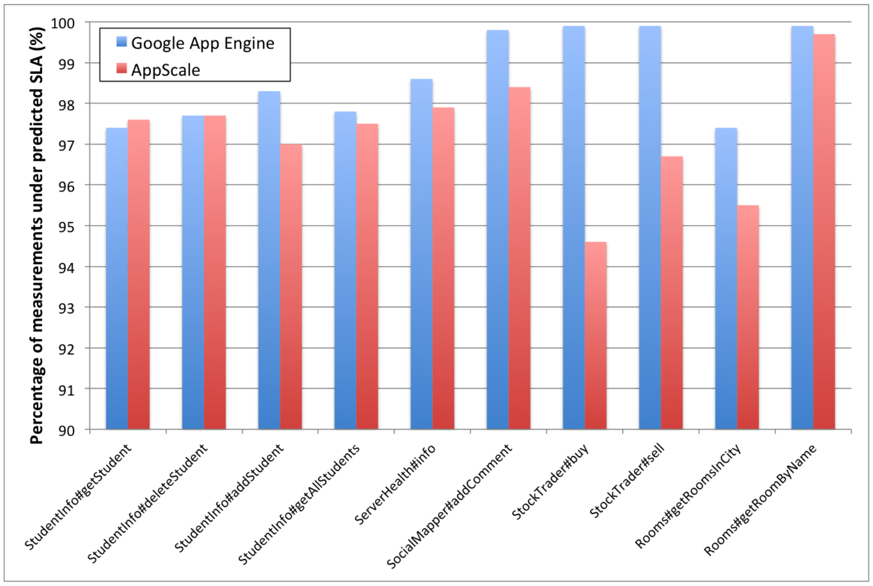

Figure 4.4: Cerebro correctness percentage in Google App Engine and AppScale cloud platforms.

istrator in production. However, having per-minute predictions enables us to compare these predictions against actual web API execution times measured during the same time period to evaluate Cerebro correctness. More specifically, we associate with each measurement the prediction from the prediction time series that most nearly precedes it in time. The correctness fraction is computed from a sample of 1000 prediction-measurement pairs.

Figure 4.4 shows the final results of this experiment. Each of the columns in figure 4.4 corresponds to a single web API operation in one of the sample applications. The columns are labeled in the form of *ApplicationName#OperationName*, a convention we will continue to use in the rest of the section. To maintain clarity in the figures we do not illustrate the results for all web API operations in the sample applications. Instead we present the results for a selected set of web API operations covering all five sample applications. We note that other web API operations we tested also produce very similar results.

Since we are using Cerebro to predict the 95^{th} percentile of the API response times, Cerebro's predictions are correct when at least 95% of the measured response times are less than their corresponding predicted upper bounds. According to figure 4.4, Cerebro achieves this goal for all the applications in both cloud environments. The lowest percentage accuracy observed in our tests is 94.6% (in the case of StockTrader#buy on AppScale), which is also very close to the target of 95%. Such minor lapses below 95% are acceptable anyway, since we expect percentage accuracy value to be gently fluctuating around some average value over time (a phenomenon that will be explained in our later results). Overall, this result shows us that Cerebro produces highly accurate SLO predictions for a variety of applications running on two very different cloud platforms.

The web API operations illustrated in Figure 4.4 cover a wide spectrum of scenarios that may be encountered in real world. StudentInfo#getStudent and StudentInfo#addStudent are by far the simplest operations in the mix. They invoke a single cloud SDK operation each, and perform a simple datastore read and a simple datastore write respectively. As per our survey results, these alone cover a significant portion of the web APIs developed for the App Engine and AppScale cloud platforms (1 path through the code, and 1 cloud SDK call). The StudentInfo#deleteStudent operation makes two cloud SDK operations in sequence, whereas StudentInfo#getAllStudents performs an iterative datastore read. In our experiment with StudentInfo#getAllStudents, we had the datastore preloaded with 1000 student records, and Cerebro was configured to use a maximum entity count of 1000 when making predictions.

ServerHealth#info invokes the same cloud SDK operation three times in sequence. Both StockTrader#buy and StockTrader#sell have multiple paths through the application (due to branching), thus causing Cerebro to make multiple sequences of predictions – one sequence per path. The results shown in Figure 4.4 are for the longest paths which consist of seven cloud SDK invocations each. According to our survey, 99.8% of the execution paths found in Google App Engine applications have seven or fewer cloud SDK calls in them. Therefore we believe that the StockTrader web API represents an important upper bound case.

Rooms#getRoomByName invokes two different cloud SDK interfaces, namely datastore and memcache. Rooms#getAllRooms is another operation that consists of an iterative datastore read. In this case, we had the datastore preloaded with 10 entities, and Cerebro was configured to use a maximum entity count of 10.

4.3.2 Tightness of Predictions

In this section we discuss the tightness of the predictions generated by Cerebro. Tightness is a measure of how closely the predictions bound the actual response times of the web APIs. Note that it is possible to perfectly achieve the correctness goal by simply predicting overly large values for web API response times. For example, if Cerebro were to predict a response time of several years for exactly 95% of the web API invocations and zero for the others, it would likely achieve a correctness percentage of 95%. From a practical perspective, however, such an extreme upper bound is not useful as an SLO.

Figure 4.5 depicts the average difference between predicted response time bounds and actual response times for our sample web APIs when running in the App Engine and AppScale clouds. These results were obtained considering a sequence of 1000 consecutive predictions (of 95^{th} percentile), and the averages are computed only for correct predictions (i.e. ones above their corresponding measurements).

According to Figure 4.5, Cerebro generates fairly tight SLO predictions for most web API operations considered in the experiments. In fact, 14 out of the 20 cases illustrated in the figure show average difference values less than 65ms. In a few cases, however, the bounds differ from the average measurement substantially:

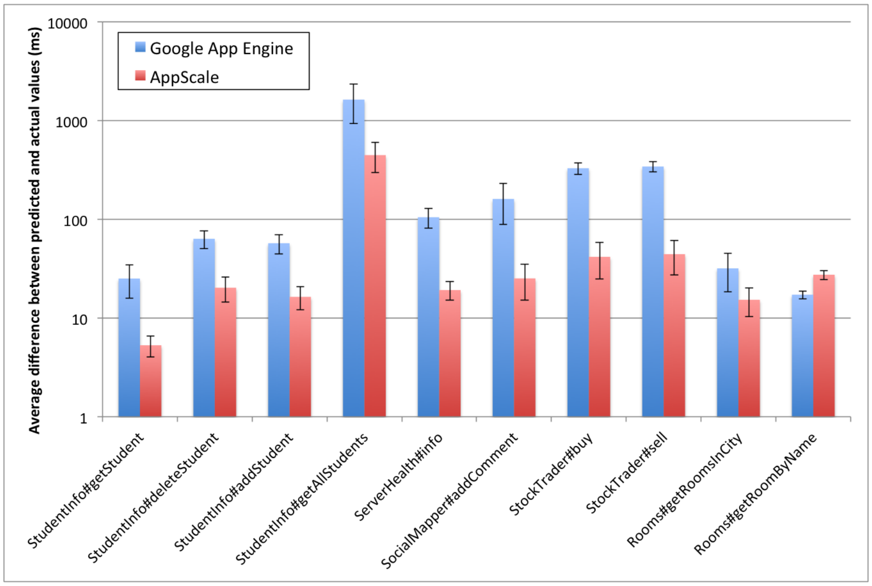

Figure 4.5: Average difference between predictions and actual response times in Google App Engine and AppScale. The y-axis is in log scale.

- StudentInfo#getAllStudents on both cloud platforms
- ServerHealth#info, SocialMapper#addComment, StockTrader#buy and StockTrader#sell on App Engine

To understand why Cerebro generates conservative predictions for some operations we further investigate the performance characteristics of them. We take StudentInfo#getAllStudents operation on App Engine as a case study, and analyze its execution time measurements in depth. This is the case which exhibits the largest average difference between predicted and actual execution times.

Figure 4.6 shows the empirical cumulative distribution function (CDF) of measured execution times for the StudentInfo#getAllStudents on Google App Engine. This distribution was obtained by considering the application's instrumentation results gathered within a window of 1000 minutes. The average of this sample is 3431.79ms, and the 95^{th} percentile from the CDF is 4739ms. Thus, taken as a distribution, the "spread" between the average and the 95^{th} percentile is more than 1300ms.

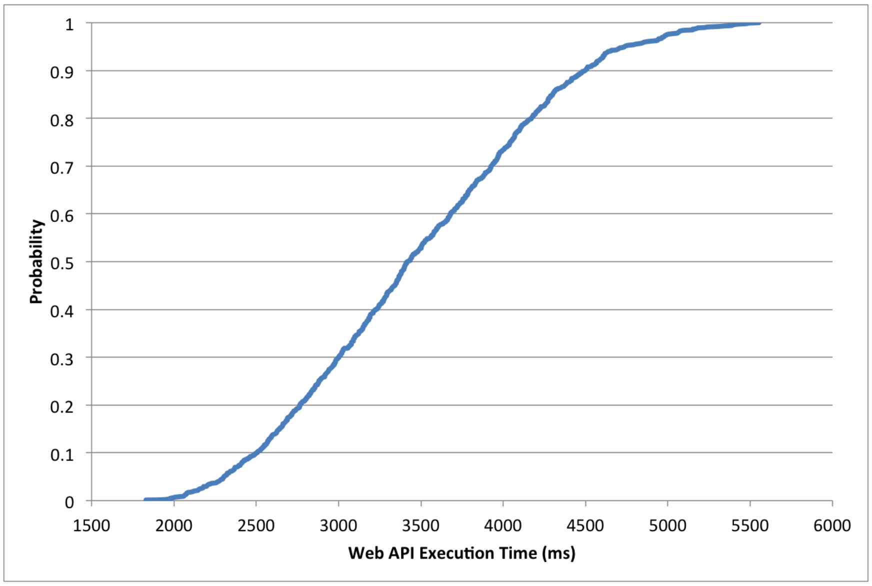

Figure 4.6: CDF of measured executions times of the StudentInfo#getAllStudents operation on App Engine.

From this, it becomes evident that StudentInfo#getAllStudents records very high execution times frequently. In order to incorporate such high outliers, Cerebro must be conservative and predict large values for the 95^{th} percentile. This is a required feature to ensure that 95% or more API invocations have execution times under the predicted SLO. But as a consequence, the average distance between the measurements and the predictions increases significantly.

We omit a similar analysis of the other cases in the interest of brevity but summarize the tightness results as indicating that Cerebro achieves a bound that is “tight” with respect to the percentiles observed by sampling the series for long periods.

Another interesting observation we can make regarding the tightness of predictions is that the predictions made in the AppScale cloud platform are significantly tighter than the ones made in Google App Engine (Figure 4.5). For nine out of the ten operations tested, Cerebro has generated tighter predictions in the AppScale environment. This is because web API performance on AppScale is far more stable and predictable thus resulting in fewer measurements that occur far from the average.

The reason why AppScale's performance is more stable over time is because it is deployed on a set of closely controlled, and monitored cluster of virtual machines (VMs) that use a private Infrastructure-as-a-Service (IaaS) cloud to implement isolation. In particular, the VMs assigned to AppScale do not share nodes with "noisy neighbors" in our test environment. In contrast, Google App Engine does not expose the performance characteristics of its multi-tenancy. While it operates at vastly greater scale, our test applications also exhibit wider variance of web API response time when using it. Cerebro, however, is able to predict a correct and tight SLOs for applications running in either platform: the lower variance private AppScale PaaS, and the extreme scale but more varying Google App Engine PaaS.

4.3.3 SLO Validity Duration

To be of practical value to PaaS administration, the duration over which a Cerebro prediction remains valid must be long enough to allow appropriate remedial action when load conditions change, and the SLO is in danger of being violated. In particular, SLOs must remain correct for at least the time necessary to allow human responses to changing conditions such as the commitment of more resources to web APIs that are in violation or alerts to support staff that customers may be calling to claim SLO breach (which likely resulted in a higher level SLA violation). Ideally, each prediction should persist as correct for several hours or more to match staff response time to potential SLO violations.

However, determining when a Cerebro-predicted SLO becomes invalid is potentially complex. For example, given the definition of correctness described in subsection 4.3.1, it is possible to report an SLO violation when the running tabulation of correctness percentage falls below the target probability (when expressed as a percentage). However, if

Table 4.2: Prediction validity period distributions of different operations in App Engine. Validity durations were computed by observing 3 consecutive SLO violations. 5^{th} and 95^{th} columns represent the 5th and 95th percentiles of the distributions respectively. All values are in hours.

Operation	5^{th}	Average	95^{th}
StudentInfo#getStudent	7.15	70.72	134.43
StudentInfo#deleteStudent	2.55	37.97	94.37
StudentInfo#addStudent	1.45	26.8	64.78
ServerHealth#info	1.41	39.22	117.71
Rooms#getRoomByName	7.24	70.47	133.36
Rooms#getRoomsInCity	2.08	30.12	82.58

this metric is used, and Cerebro is correct for many consecutive measurements, a sudden change in conditions that causes the response time to persist at a higher level will not immediately trigger a violation. For example, Cerebro might be correct for several consecutive months and then incorrect for several consecutive days before the overall correctness percentage drops below 95%, and a violation is detected. If the SLO is measured over a year, such time scales may be acceptable but we believe that PaaS administrators would consider such a long period of time where the SLOs were continuously in violation unacceptable. Thus we adopt the more conservative approach described in section 4.2.5 to measure the duration over which a prediction remains valid than simply measuring the time until the correctness percentage drops below the SLO-specified value. Tables 4.2 and 4.3 present these durations for Cerebro predictions in Google App Engine and AppScale respectively. These results were calculated by analyzing a trace of data collected over 7 days.

From Table 4.2 the average validity duration for all 6 operations considered in App Engine is longer than 24 hours. The lowest average value observed is 26.8 hours, and that is for the StudentInfo#addStudent operation. If we just consider the 5^{th} percentiles of the distributions, they are also longer than 1 hour. The smallest 5^{th} percentile value of

Table 4.3: Prediction validity period distributions of different operations in AppScale. Validity periods were computed by observing 3 consecutive SLO violations. 5^{th} and 95^{th} columns represent the 5th and 95th percentiles of the distributions respectively. All values are in hours.

Operation	5^{th}	Average	95^{th}
StudentInfo#getStudent	6.1	60.67	115.24
StudentInfo#deleteStudent	6.08	60.21	114.32
StudentInfo#addStudent	6.1	60.67	115.24
ServerHealth#info	6.29	54.53	108.14
Rooms#getRoomByName	6.07	59.18	112.28
Rooms#getRoomsInCity	1.95	33.77	84.63

1.41 hours is given by the ServerHealth#info operation. This result implies that, based on our conservative model for detecting SLO violations, Cerebro predictions made on Google App Engine would be valid for at least 1.41 hours or more, at least 95% of the time.

By comparing the distributions for different operations we can conclude that API operations that perform a single basic datastore or memcache read tend to have longer validity durations. In other words, those cloud SDK operations have fairly stable performance characteristics in Google App Engine. This is reflected in the 5^{th} percentiles of StudentInfo#getStudent and Rooms#getRoomByName. Alternatively operations that execute writes, iterative datastore reads or long sequences of cloud SDK operations have shorter prediction validity durations.

For AppScale, the smallest average validity duration of 33.77 hours is observed from the Rooms#getRoomsInCity operation. All other operations tested in AppScale have average prediction validity durations greater than 54 hours. The lowest 5^{th} percentile value in the distributions, which is 1.95 hours, is also shown by Rooms#getRoomsInCity. This means, the SLOs predicted for AppScale would hold correct for at least 1.95 hours or more, at least 95% of the time. The relatively smaller validity durations values computed for the Rooms#getRoomsInCity operation indicates that the performance of iterative datastore reads is subject to some variability in AppScale.

4.3.4 Long-term SLO Durability and Change Frequency

In this section we further analyze how the Cerebro-predicted SLOs change over long periods of time (e.g. several months). Our goal is to understand the frequency with which Cerebro's auto-generated SLOs get updated due to the changes that occur in the cloud platform, and the time duration between these update events. That is, we assess the number of times an API consumer is prompted with an updated SLO, thereby potentially initiating SLA renegotiations.

To enable this, we deploy Cerebro's cloud SDK monitoring agent in the Google App Engine cloud, and benchmark the cloud SDK operations every 60 seconds for 112 days. We then use Cerebro to make SLO predictions (95th percentile) for a set of test web applications. Note that we conduct this long-term experiment only on App Engine, which according to our previous results gives shorter SLO validity durations than AppScale.

Cerebro analyzes the benchmarking results collected by the cloud SDK monitor, and generates sequences of SLO predictions for the web APIs of each application. Each prediction sequence is a time series that spans the duration in which the cloud SDK monitor was active in the cloud. Each prediction is timestamped. Therefore given any timestamp that falls within the 112 day period of the experiment, we can find an SLO prediction that is closest to it. Further, we associate each prediction with an integer value (C_w) which indicates the consecutive number of SLO violations that should be observed, before we may consider the prediction to be invalid.

We also estimate the actual web API response times for the test applications. This is done by simply summing up the benchmarking data gathered by the cloud SDK monitor.

Again, we assume that the time spent on non cloud SDK operations is negligible. For example, consider a web API that executes the cloud SDK operations O_1, O_2 and O_1 in that order. Now suppose the cloud SDK monitor has gathered following benchmarking results for O_1 and O_2:

- O_1: $[t_1 : x_1,\, t_2 : x_2,\, t_3 : x_3 ...]$
- O_2: $[t_1 : y_1,\, t_2 : y_2,\, t_3 : y_3 ...]$

Here t_i are timestamps at which the benchmark operations were performed. x_i and y_i are execution times of the two SDK operations measured in milliseconds. Given this benchmarking data, we can calculate the time series of actual response time of the API as follows:

$[t_1 : 2x_1 + y_1,\, t_2 : 2x_2 + y_2,\, t_3 : 2x_3 + y_3 ...]$

The coefficient 2 that appears with each x_i term accounts for the fact that our web API invokes O_1 twice. In this manner, we can combine the static analysis results of Cerebro with the cloud SDK benchmarking data to obtain a time series of estimated actual response times for all web APIs in our sample applications.

Having obtained a time series of SLO predictions (T_p) and a time series of actual response times (T_a) for each web API, we perform the following computation. From T_p we pick a pair $< s_0, t_0 >$, where s_0 is a predicted SLO value and t_0 is the timestamp associated with it. Then starting from t_0, we scan the time series T_a to detect the earliest point in time at which we can consider the predicted SLO value s_0 as invalid. This is done by comparing s_0 against each entry in T_a that has a timestamp greater than or equal to t_0, until we see C_w consecutive entries that are larger than s_0. Here C_w is the rare event threshold computed by Cerebro when making SLO predictions. Having found such an SLO invalidation event at time t', we record the duration $t' - t_0$ (i.e. the SLO validity duration), and increment the counter *invalidations*, which starts from 0. Then we pick the pair $< s_1, t_1 >$ from T_p where t_1 is the smallest timestamp greater than or equal to t', and s_1 is the predicted SLO value at that timestamp. Then we scan T_a starting from t_1, until we detect the next SLO invalidation (for s_1). We repeat this process until we exhaust either T_p or T_a. At the end of this computation we have a distribution of SLO validity periods, and the counter *invalidations* indicates the number of SLO invalidations we encountered in the process.

This experimental process simulates how a single API consumer encounters SLO changes. Selecting the first pair of values $< s_0, t_0 >$ represents the API consumer receiving an SLO for the first time (i.e. at API subscription). When this SLO becomes invalid, the API consumer receives a new SLO, which is represented by the selection of the pair $< s_1, t_1 >$. Therefore, when the simulation reaches the end of the time series, we can determine how many times the API consumer observed changes to the SLO (given by *invalidations*). The recorded SLO validity periods give an indication of the time between these SLO change events.

For a given web API we perform the above simulation many times, using each entry in T_p as a starting point. That is, in each run we change our selection of $< s_0, t_0 >$ to be a different entry in T_p. This way, for a time series comprised of n entries, we can run the simulation $n-1$ times, discarding the last entry. We can assume that each simulation run corresponds to a different API consumer. Therefore, at the end of a complete execution of the experiment we have the number of SLO changes for many different API consumers, and the empirical SLO validity period distributions for each of them.

The smallest n we encountered in all our experiments was 125805. That is, we repeatedly simulated each web API SLO trace for at least 125804 API consumers. Similarly, the largest number of API consumers we performed the simulation for is 145130.

We now present the experimental results obtained using this methodology. We analyze the number of SLO changes observed by each API consumer during the 112 day

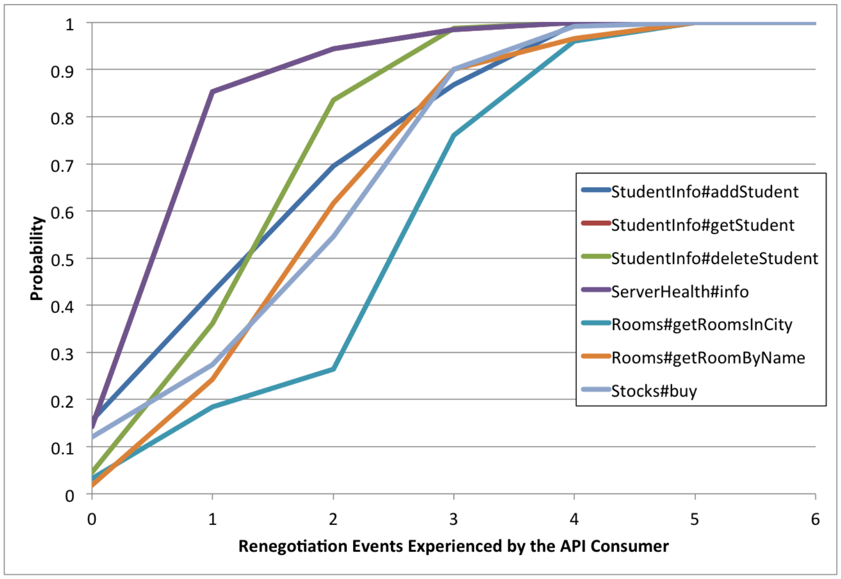

Figure 4.7: CDF of the number of SLO change events faced by API consumers.

period of the experiment, and calculate a set of cumulative distribution functions (CDF). These CDFs describe the probability of finding an API consumer that experienced a given number of SLO change events. Figure 4.7 presents the CDFs. We use the convention *ApplicationName#Operation* to label individual web API operations.

According to Figure 4.7, the largest number of SLO changes experienced by any user is 6. This is with regard to the StudentInfo#addStudent operation. Across all web APIs, at least 96% of the API consumers experience no more than 4 SLO changes during the period of 112 days. Further, at least 76% of the API consumers see no more than 3 SLO changes. These statistics indicate that SLOs predicted by Cerebro for Google App Engine are stable over time, and reassessment is required only rarely. From an API consumer's perspective this is a highly desirable property, since it reduces the frequency of SLO changes, which reduces the potential SLA renegotiation overhead.

Next we analyze the time duration between SLO change events. For this we combine the SLO validity periods computed for different API consumers into a single statistical distribution. Table 4.4 shows the 5th percentile, mean, and 95th percentile of these

Operation	5^{th}	Mean	95^{th}
StudentInfo#getStudent	12.97	631.24	1911.19
StudentInfo#deleteStudent	7.65	472.07	2031.59
StudentInfo#addStudent	0.05	458.24	1711.08
ServerHealth#info	12.96	630.01	1911.19
Rooms#getRoomByName	8.48	345.13	1096.53
Rooms#getRoomsInCity	20.56	296.44	1143.45
Stocks#buy	8.46	411.75	815.5

Table 4.4: Prediction validity period distributions (in hours). 5^{th} and 95^{th} columns represent the 5th and 95th percentiles of the distributions respectively.

combined distributions.

The smallest mean SLO validity period observed in our experiments is 296.44 hours (12.35 days). This value is given by the Rooms#getRoomsInCity operation. This implies that on average, API consumers do not see a change in Cerebro-predicted SLOs for at least 12.35 days. Similarly, we observed the largest mean SLO validity period of 26.3 days with the StudentInfo#getStudent operation. The smallest 5th percentile value of 0.05 hours is shown by the StudentInfo#addStudent operation, but this appears to be a special case compared to the other web API operations. The second smallest 5th percentile value of 7.65 hours is shown by the StudentInfo#deleteStudent operation. Therefore, ignoring the StudentInfo#addStudent operation, API consumers observe SLO validity periods longer than 7.65 hours at least 95% of the time. That is, the time between SLO changes is greater than 7.65 hours at least 95% of the time.

To reduce the number of SLO changes further, we observe that we can exploit the SLO change events in which the difference between an invalidated SLO and a new SLO is small. In such cases, it is of little use to provide a new SLO, and API consumers may be content to continue with the old SLO. To incorporate this behavior into Cerebro (and our simulation process), we introduce threshold value *slo_delta_threshold* into the process. This parameter takes a percentage value that represents the minimum acceptable

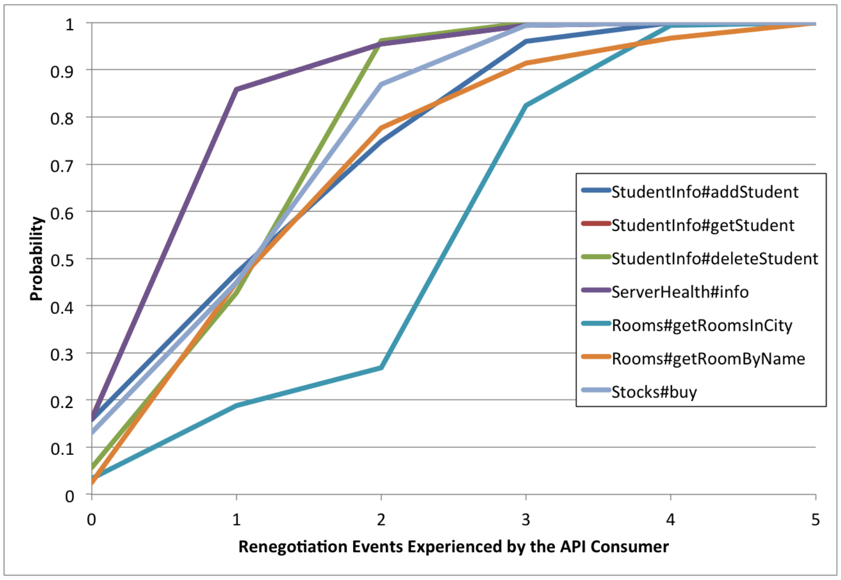

Figure 4.8: CDF of the number of SLO change events faced by API consumers, when *slo_delta_threshold* = 10%

percentage difference between the old and new SLO values before renegotiation. If the percentage difference between the two SLO values is below this threshold, we do not record the SLO validity period, nor increment the count of the SLO invalidations. That is, we do not consider such cases as SLO change events. We simply carry on with the old SLO value until we come across an invalidation event with a percentage difference that exceeds the threshold. Note that our previous experiments are a special case of thresholding for which *slo_delta_threshold* is 0.

Next we evaluate the sensitivity of our results to *slo_delta_threshold*. Figure 4.8 shows the resulting CDFs of per-user renegotiation count when the threshold is 10%. That is, Cerebro does not prompt the API consumer with an SLO change, unless the new SLO is at least 10% off from the old one. In this case, the maximum number of SLO change events drops from 6 to 5. Also most of the probabilities shift slightly upwards. For instance, now more than 82% of the users see 3 or less renegotiation events (as opposed to 76%).

Operation	5^{th}	Mean	95^{th}
StudentInfo#getStudent	19.93	644.58	1911.19
StudentInfo#deleteStudent	7.93	512.52	2031.59
StudentInfo#addStudent	0.05	491.68	1711.08
ServerHealth#info	19.91	643.33	1911.19
Rooms#getRoomByName	8.48	392.01	1096.53
Rooms#getRoomsInCity	21.82	304.97	1143.45
Stocks#buy	7.41	510.31	1277.7

Table 4.5: Prediction validity period distributions (in hours) when *slo_delta_threshold* = 10%. 5^{th} and 95^{th} columns represent the 5th and 95th percentiles of the distributions respectively.

Table 4.5 shows the SLO validity period distributions computed when *slo_delta_threshold* is 10%. Here, as expected most of the mean and 5th percentile values have increased slightly from their original values. The smallest mean value recorded in the table is 304.97 hours. We have also considered a *slo_delta_threshold* value of 20%. This change introduces only small shifts in the probability values of the CDFs (more than 84% of the users see 3 or less renegotiations), and the maximum number of renegotiations remains at 5.

In summary, we find that the performance SLOs predicted by Cerebro for the Google App Engine cloud environment are stable over time. That is, the predictions are valid for long periods of time, and API consumers do not observe SLO changes often. In our experiment spanning over a period of 112 days, the maximum number of SLO changes a user had to undergo was 6. More than 76% of the users experienced only 3 or less changes. We can further reduce the number of SLO changes per API consumer by introducing a threshold for the minimum applicable percentage SLO change. This helps to eliminate the cases where an old SLO has been marked as invalid by our statistical model for detecting SLO invalidations, but the new SLO predicted by Cerebro is not very different from the old one. However, the effect of this parameter starts to diminish as we increase its value. In our experiments, we observe the best results for a threshold of 10%. Using a value of 20% does not achieve significantly better results.

4.3.5 Effectiveness of QBETS

In order to gauge the effectiveness of QBETS, we compare it to a “naïve” approach that simply uses the running empirical percentile tabulation of a given joint time series as a prediction. This *simple predictor* retains a sorted list of previous observations, and predicts the p-th percentile to be the value that is larger than p% of the values in the observation history. Whenever a new observation is available, it is added to the history and each prediction uses the full history.

Figure 4.9 shows the correctness measurements for the simple predictor using the same cloud SDK monitoring data and application benchmarking data that was used in Subsection 4.3.1. That is, we keep the rest of Cerebro unchanged, swap QBETS out for the simple predictor, and run the same set of experiments using the logged observations. Thus the results in figure 4.9 are directly comparable to figure 4.4 where Cerebro uses QBETS as a forecaster.

For the simple predictor, Figure 4.9 shows lower correctness percentages compared to Figure 4.4 for QBETS (i.e. the simple predictor is less conservative). However, in several cases the simple predictor falls well short of the target correctness of 95% necessary for the SLO. That is, it is unable to furnish a prediction correctness that can be used as the basis of an SLO in all of the test cases. This indicates that QBETS is a superior approach, albeit conservativeness, for making SLO predictions than simply calculating the percentiles on cloud SDK monitoring data.

To illustrate why the simple predictor fails to meet the desired correctness level, figure 4.10 shows the time series of observations, simple predictor forecasts, and QBETS

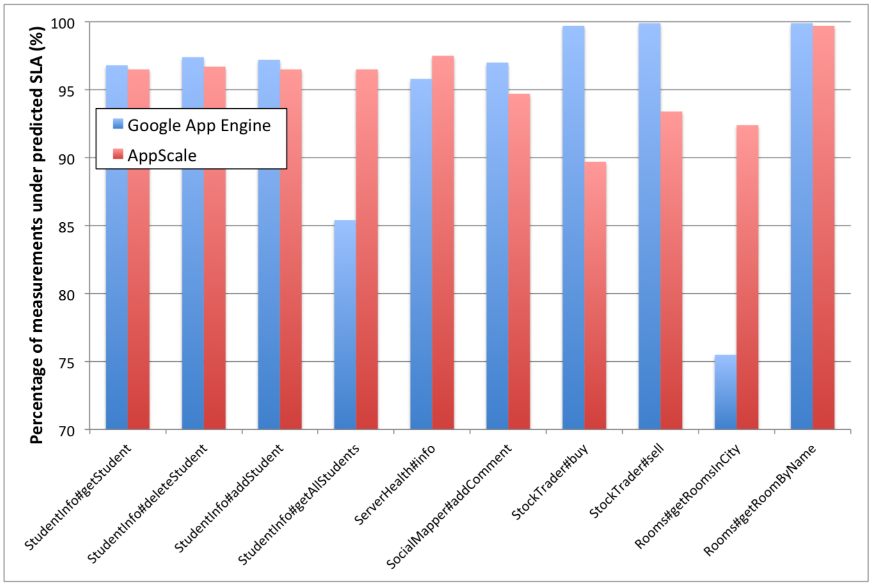

Figure 4.9: Cerebro correctness percentage resulting from the simple predictor (without QBETS).

forecasts for the Rooms#getRoomsInCity operation on Google App Engine (the case in figure 4.9 that shows lowest correctness percentage).

In this experiment, there are a significant number of response time measurements that violate the SLO given by simple predictor (i.e. are larger than the predicted percentile), but are below the corresponding QBETS prediction made for the same observation. Notice also that while QBETS is more conservative (its predictions are generally larger than those made by the simple predictor), in this case the predictions are typically only 10% larger. That is, while the simple predictor shows the 95^{th} percentile to be approximately $40ms$, the QBETS predictions vary between $42ms$ and $48ms$, except at the beginning where QBETS is "learning" the series. This difference in prediction, however, results in a large difference in correctness percentage. For QBETS, the correctness percentage is 97.4% (Figure 4.4) compared to 75.5% for the simple predictor (Figure 4.9).

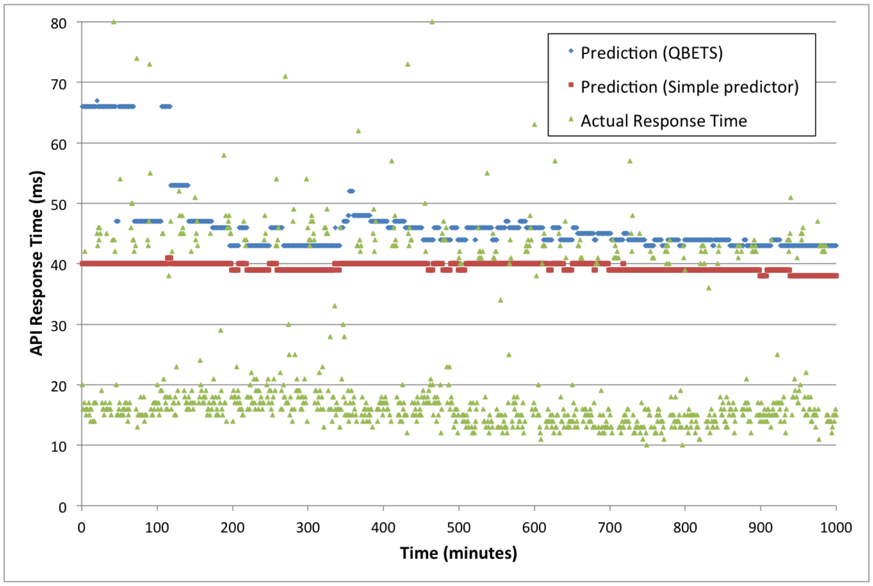

Figure 4.10: Comparison of predicted and actual response times of Rooms#getRoomsInCity on Google App Engine.

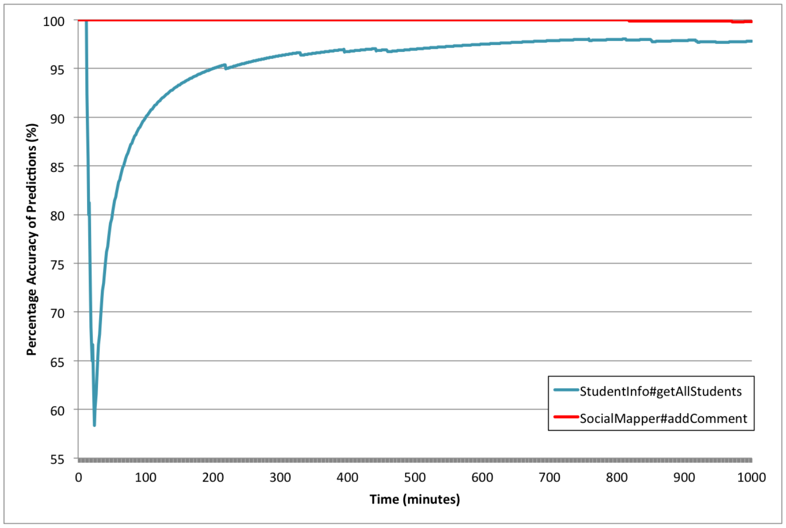

Figure 4.11: Running tabulation of correctness percentage for predictions made on App Engine for a period of 1000 minutes, one prediction per minute.

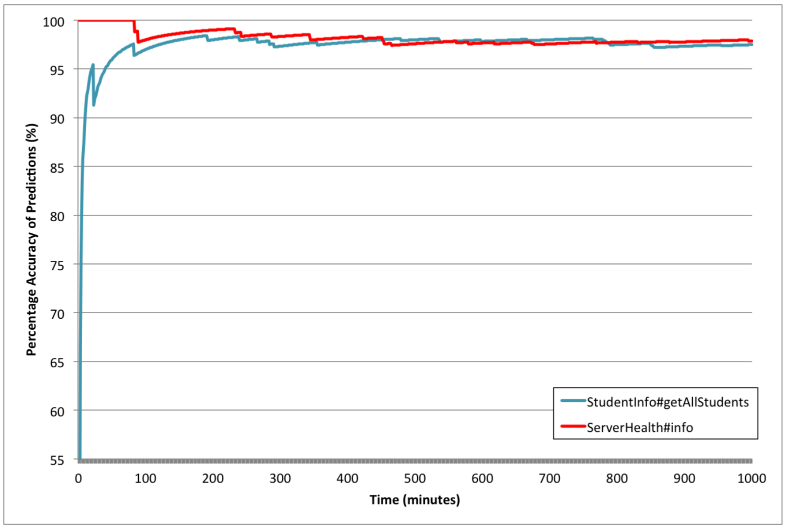

Figure 4.12: Running tabulation of correctness percentage for predictions made on AppScale for a period of 1000 minutes, one prediction per minute.

4.3.6 Learning Duration

As described in subsection 4.2.3, QBETS uses a form of supervised learning internally to determine each of its bound predictions. Each time a new prediction is presented, it updates its internal state with respect to autocorrelation and change-point detection. As a result, the correctness percentage may require some number of state updates to converge to a stable value.

Figure 4.11 shows a running tabulation of correctness percentage for Cerebro predictions made in Google App Engine during the first 1000 minutes of operation (one prediction is generated each minute). Similarly, in figure 4.12 we show a running tabulation of correctness percentage for Cerebro predictions made in AppScale during the first 1000 minutes of operation (again, one prediction generated per minute).

For clarity we do not show results for all tested operations. Instead, we only show data for the operation that reaches stability in the shortest amount of time, and the operation that takes the longest to converge. Results for other operations fall between these two extremes.

In the worst case, Cerebro takes up to 200 minutes to achieve correctness percentage above 95% in Google App Engine (for StudentInfo#getAllStudents). Alternatively, the longest time until Cerebro has "learned" the series in AppScale is approximately 40 minutes.

Summarizing these results, the learning time for Cerebro may be several hours (up to 200 minutes in case of Google App Engine), before it produces trustworthy and correct SLO predictions. The predictions made during this learning period are not necessarily incorrect. It is just not possible to gauge their correctness quantitatively before the series has been learned. We envision Cerebro as a continuous monitoring process in PaaS clouds for which "startup time" is not an issue.

4.4 Related Work

Our research leverages a number of mature research areas in computer science and mathematics. These areas include static program analysis, cloud computing, time series analysis, and SOA governance.

The problem of predicting response time SLOs of web APIs is similar to worst-case execution time (WCET) analysis [107, 108, 109, 100, 110]. The objective of WCET analysis is to determine the maximum execution time of a software component in a given hardware platform. It is typically discussed in the context of real-time systems, where the developers should be able to document and enforce precise hard real-time constraints on the execution time of programs. In order to save time, manpower and hardware resources, WCET analysis solutions are generally designed favoring static analysis methods over software testing. We share similar concerns with regard to cloud platforms, and strive to eliminate software testing in the favor of static analysis.

Ermedahl et al describe SWEET [108], a WCET analysis tool that make use of program slicing [109], abstract interpretation [111], and invariant analysis [100] to determine the loop bounds and worst-case execution time of a program. Program slicing used in this prior work to limit the amount of code being analyzed is similar in its goal to our focus on cloud SDK invocations. SWEET uses abstract interpretation in interval and congruence domains to identify the set of values that can be assigned to key control variables of a program. These sets are then used to calculate exact loop bounds for most data-independent loops in the code. Invariant analysis is used to detect variables that do not change during the course of a loop iteration, and remove them from the analysis thus further simplifying the loop bound estimation. Lokuceijewski et al propose a similar WCET analysis using program slicing and abstract interpretation [112]. They additionally use a technique called polytope models to speed up the analysis.

The corpus of research that covers the use of static analysis methods to estimate the execution time of software applications is extensive. Gulwani, Jain and Koskinen used two techniques named control-flow refinement and progress invariants to estimate the bounds for procedures with nested and multi-path loops [113]. Gulwani, Mehra and Chilimbi proposed SPEED [114], a system that computes symbolic bounds for programs. This system makes use of user-defined quantitative functions to predict the bounds for loops iterating over data structures like lists, trees and vectors. Our idea of using user-defined values to bound data-dependent loops (e.g. iterative datastore reads) is partly inspired by this concept. Bygde [101] proposed a set of algorithms for predicting data-independent loops using abstract interpretation and element counting (a technique that was partly used in [108]). Cerebro incorporates minor variations of these algorithms successfully due to their simplicity.

Cerebro makes use of and is similar to many of the execution time analysis systems discussed above. However, there are also several key differences. For instance, Cerebro is focused on solving the execution time prediction problem for PaaS-hosted web APIs. As we show in our characterization survey, such applications have a set of unique properties, that can be used to greatly simplify static analysis. Also, Cerebro is designed to only work with web API codes. This makes designing a solution much more simpler but less general. To handle the highly variable and evolving nature of cloud platforms, Cerebro combines static analysis with runtime monitoring of cloud platforms at the level of SDK operations. No other system provides such a hybrid approach to the best of our knowledge. Finally, we use time series analysis [90] to predict API execution time upper bounds with specific confidence levels.

SLA management on service-oriented systems and cloud systems has been throughly researched over the years. However, a lot of the existing work has focused on issues such as SLA monitoring [115, 116, 117, 118], SLA negotiation [119, 120, 121], and SLA modeling [122, 123, 124]. Some work has looked at incorporating a given SLA to the design of a system, and then monitoring it at the runtime to ensure SLA compliant behavior [125]. Our research takes a different approach from such works, whereby it attempts to predict the performance SLOs for a given web API, which in turns can be used to formulate performance SLAs between API providers and consumers. To the best of our knowledge, Cerebro is the first system to predict performance SLOs for web APIs developed for PaaS clouds.

A work that is similar to ours has been proposed by Ardagna, Damiani and Sagbo in [126]. The authors develop a system for early estimation of service performance based on simulations. Given a STS model (Symbolic Transitions System) of a service, their system is able to generate a simulation script, which can be used to assess the performance of the service. STS models are a type of finite state automata. Further, they use probabilistic distributions with fixed parameters to represent the delays incurred by various operations in the service. Cerebro is easier to use than this system because we do not require API developers to construct any models of the web APIs. They only need to provide the source code of the API implementations. Also, instead of using probabilistic distributions with fixed parameters, Cerebro uses actual historical performance metrics of cloud SDK operations. This enables Cerebro to generate more accurate results, that reflect the dynamic nature of the cloud platform.

In PROSDIN [119], a proactive service discovery and negotiation framework, the SLA negotiation occurs during the service discovery phase. This is similar to how Cerebro provides an initial SLO with an API consumer, when the consumer subscribes to an API. PROSDIN also establishes a fixed SLA validity period upon negotiation, and triggers an SLA renegotiation when this time period has elapsed. Cerebro on the other hand continuously monitors the cloud platform, and periodically re-evaluates the response time SLOs of web APIs to determine when a reassessment is needed. Similarly, researchers have investigated the notions of SLA brokering [121], and the automatic SLA negotiation between intelligent agents [120], ideas that can complement the simple SLO provisioning model of Cerebro to make it more powerful and flexible.

Meryn [127] is an SLA-driven PaaS system that attempts to maximize cloud provider profit, while providing the best possible quality of service to the cloud users. It supports SLA negotiation at application deployment, and SLA monitoring to detect violations. However, it does not automatically determine what SLAs are feasible or address SLA renegotiation, and employs a policy-based mechanism coupled with a penalty cost charged against the cloud provider to handle SLA violations. Also, Meryn formulates SLAs in terms of the computing resources (CPU, memory, storage etc.) allocated to applications. It assumes a batch processing environment where the execution time of an application is approximated based on a detailed description of the application provided by the developer. In contrast, Cerebro handles SLOs for interactive web applications. It predicts the response time of applications using static analysis, without any input from the application developer. Cerebro also supports automatic SLO reassessment, with possible room for economic incentives.

Iosup et al showed via empirical analysis, that production cloud platforms like Google App Engine and AWS regularly undergo performance variations, thus impacting the response time of the applications deployed in such cloud platforms [128]. Some of these cloud platforms even exhibit temporal patterns in their performance variations (weekly, monthly, annual or seasonal). Cerebro and the associated API performance forecasting model acknowledge this fact, and periodically reassess the predicted response time upper bounds. It detects when a previously predicted upper bound becomes invalid, and prompts the API clients to update their SLOs accordingly. Indeed, one of Cerebro's strength's is its ability to detect change points in the input time series data (periodically collected cloud SDK benchmark results), and generate up-to-date predictions that are not affected by old obsolete observations that were gathered prior to a change point.

There has also been prior work in the area of predicting SLO violations [129, 130, 131]. These systems take an existing SLO and historical performance data of a service, and predict when the service might violate the given SLO in the future. Cerebro's notion of SLO validity period has some relation to this line of research. However, Cerebro's main goal is to make SLO predictions for web APIs *before* they are deployed and executed. We believe that some of these existing SLO violation predictors can complement our work by providing API developers and cloud administrators insights on when a Cerebro-predicted SLO will be violated.

4.5 Conclusions and Future Work

Stipulating SLOs (bounds) on the response time of web APIs is crucial for implementing several features related to automated governance. To this end we present Cerebro, a system that predicts response time SLOs for web APIs deployed in PaaS clouds. The

SLOs predicted by Cerebro can be used to enforce policies regarding the performance level expected from cloud-hosted web applications. They can be used to negotiate SLAs with API clients. They can also be used as thresholds when implementing application performance monitoring (APM) – subject of the next chapter. Cerebro is intended for use during development and deployment phases of a web API, and precludes the need for continuous performance testing of the API code. Further, it does not interfere with run-time operation (i.e. it requires no application instrumentation) making it scalable.

Cerebro uses static analysis to extract the sequence of cloud SDK calls (i.e. PaaS kernel invocations) made by a given web API code, and combines that with the historical performance measurements of individual cloud SDK calls. Cerebro employs QBETS, a non-parametric time series analysis and forecasting method, to analyze cloud SDK performance data, and predict bounds on API response time that can be used as statistical "guarantees" with associated guarantee probabilities.

We have implemented a prototype of Cerebro for Google App Engine public PaaS, and AppScale private PaaS. We evaluate it using a set of representative and open source web applications developed by others. Our findings indicate that the prototype can determine response time SLOs with target accuracy levels specified by an administrator. Specifically, we use Cerebro to predict the 95th percentile of the API response time. We find that:

- Cerebro achieves the desired correctness goal of 95% for all the applications in both cloud environments.

- Cerebro generates tight predictions (i.e. the predictions are similar to measured values) for most web APIs. Because some operations and PaaS systems exhibit more variability in cloud SDK response time, Cerebro must be conservative in some cases, and produce predictions that are less tight to meet its correctness guarantees.

- Cerebro requires a “warm up” period of up to 200 minutes to produce trustworthy predictions. Since PaaS systems are designed to run continuously, this is not an issue in practice.

- We can use a simple yet administratively useful model to identify when an SLO becomes invalid to compute prediction validity durations for Cerebro. The average duration of a valid Cerebro prediction is between 24 and 72 hours, and 95% of the time this duration is at least 1.41 hours for App Engine and 1.95 hours for AppScale.

We also find that, when using Cerebro to establish SLOs, the API consumers do not experience SLO changes often, and the maximum number of times an API consumer encounters an SLO change over a period of 112 days is six. Overall, this work shows that automatic stipulation of response-time SLOs for web APIs is practically viable in real world cloud settings, and API consumer timeframes.

In the current design, Cerebro’s cloud SDK monitoring agent only monitors a predefined set of cloud SDK operations. In our future work we wish to explore the possibility of making this component more dynamic, so that it automatically learns what operations to benchmark from the web APIs deployed in the cloud. This also includes learning the size and the form of the datasets that cloud SDK invocations operate on, so that Cerebro can acquire more realistic benchmarking data. We also plan to investigate further how to better handle data-dependent loops (iterative datastore reads) for different workloads. We are interested in exploring the ways in which we can handle API codes with unpredictable execution patterns (e.g. loops based on a random number), even though such cases are quite rare in the applications we have looked at so far. Further, we plan to integrate Cerebro with EAGER, our API governance system and policy engine for PaaS clouds, so that PaaS administrators can enforce SLO-related policies on web APIs at deployment-time. Such a system will make it possible to prevent any API that does not adhere to the organizational performance standards from being deployed in the production cloud environment. It can also enforce policies that prevent applications from taking dependencies on APIs that are not up to the expected performance standards.

Chapter 5

Performance Anomaly Detection and Root Cause Analysis for Cloud-hosted Web Applications

In the previous chapter we presented a methodology for stipulating performance SLOs for cloud-hosted web applications. In this chapter we discuss detecting performance SLO violations, and conducting root cause analysis. Timely detection of performance problems, and the ability to diagnose the root causes of such issues are critical elements of governance.

This widespread adoption of cloud computing, particularly for deploying web applications, is facilitated by ever-deepening software abstractions. These abstractions elide the complexity necessary to enable scale, while making application development easier and faster. But they also obscure the runtime details of cloud applications, 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 [132].

Application developers and cloud administrators generally wish to monitor application performance, detect anomalies, and identify bottlenecks. To obtain this level of operational insight into cloud-hosted applications, and facilitate governance, the cloud platforms must support data gathering and analysis capabilities that span the entire software stack of the cloud. However, most cloud technologies available today do not provide adequate application monitoring support. Cloud administrators must therefore trust the application developers to implement necessary instrumentation at the application level. This typically entails using third party, external monitoring software [12, 13, 14], which significantly increases the effort and financial cost of maintaining applications. Developers must also ensure that their instrumentation is both correct, and does not degrade application performance. Nevertheless, since the applications depend on extant cloud services (e.g. scalable database services, scalable in-memory caching, etc.) that are performance opaque, it is often difficult, if not impossible to diagnose the root cause of a performance problem using such extrinsic forms of monitoring.

Further compounding the performance diagnosis problem, today's cloud platforms are very large and complex [132, 133]. 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 layer or the component of the cloud platform that may be responsible for it. Facilitating this level of comprehensive root cause analysis requires both data collection at different layers of the cloud, and mechanisms for correlating the events recorded at different layers.

Moreover, performance monitoring for cloud applications needs to be highly customizable. Different applications have different monitoring requirements in terms of data gathering frequency (sampling rate), length of the history to consider when performing statistical analysis (sample size), and the performance SLOs (service level objectives [89]) and policies that govern the application. Cloud monitoring should be able to facilitate these diverse requirements on a per-application basis. Designing such customizable and extensible performance monitoring frameworks that are built into the cloud platforms is a novel and challenging undertaking.

To address these needs, we present a full-stack application performance monitor (APM) called *Roots* that can be integrated with a variety of cloud Platform-as-a-Service (PaaS) technologies. PaaS clouds provide a set of managed services, which developers compose into applications. To be able to correlate application activity with cloud platform events, we design Roots as another managed service built into the PaaS cloud. Therefore it operates at the same level as the other services offered by the cloud platform. This way Roots can collect data directly from the internal service implementations of the cloud platform, thus gaining full visibility into all the inner workings of an application. It also enables Roots to operate fully automatically in the background, without requiring instrumentation of application code.

Previous work has outlined several key requirements that need to be considered when designing a cloud monitoring system [132, 133]. We incorporate many of these features into our design:

Scalability Roots is lightweight, and does not cause any noticeable overhead in application performance. It puts strict upper bounds on the data kept in memory. The persistent data is accessed on demand, and can be removed after their usefulness has expired.

Multitenancy Roots facilitates configuring monitoring policies at the granularity of individual applications. Users can employ different statistical analysis methods to process the monitoring data in ways that are most suitable for their applications.

Complex application architecture Roots collects data from the entire cloud stack (load balancers, app servers, built-in PaaS services etc.). It correlates data gathered from different parts of the cloud platform, and performs systemwide bottleneck identification.

Dynamic resource management Cloud platforms are dynamic in terms of their magnitude and topology. Roots captures performance events of applications by augmenting the key components of the cloud platform. When new processes/components become active in the cloud platform, they inherit the same augmentations, and start reporting to Roots automatically.

Autonomy Roots detects performance anomalies online without manual intervention. When Roots detects a problem, it attempts to automatically identify the root cause by analyzing available workload and service invocation data.

Roots collects most of the data it requires by direct integration with various internal components of the cloud platform. In addition to high-level metrics like request throughput and latency, Roots also records the internal PaaS service invocations made by applications, and the latency of those calls. It uses batch operations and asynchronous communication to record events in a manner that does not substantively increase request latency.

The previous two chapters present systems that perform the *specification* (policies and SLOs) and *enforcement* functions of governance. Roots also performs an important function associated with automated governance – *monitoring*. It is designed to monitor cloud-hosted web applications for SLO violations, and any other deviations from specified or expected behavior. Roots flags such issues as anomalies, and notifies cloud administrators in near real time. Also, when Roots detects an anomaly in an application, it attempts to uncover the root cause of the anomaly by analyzing the workload data, and the performance of the internal PaaS services the application depends on. Roots can determine 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 this end we propose a statistical bottleneck identification method for PaaS clouds. It uses a combination of quantile analysis, change point detection and linear regression to perform root cause analysis.

Using Roots we also devise a mechanism to identify different paths of execution in an application – i.e. different paths in the application's control flow graph. Our approach does not require static analysis, and instead uses the runtime data collected by Roots. This mechanism also calculates the proportion of user requests processed by each path, which is used to characterize the workload of an application (e.g. read-heavy vs write-heavy workload in a data management application). Based on that, Roots monitors for characteristic changes in the application workload.

We build a working prototype of Roots using the AppScale [7] open source PaaS. We evaluate the feasibility and the efficacy of Roots by conducting a series of empirical trials using our prototype. We also show that our approach for identifying performance bottlenecks in PaaS clouds, produces accurate results nearly 100% of the time. We also demonstrate that Roots does not add a significant performance overhead to the applications, and that it scales well to monitor tens of thousands of applications concurrently.

We discuss the following contributions in this chapter:

- We describe the architecture of Roots as an intrinsic PaaS service, which works automatically without requiring or 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 application component that is responsible for the anomaly.
- We present a mechanism for identifying the execution paths of an application via the runtime data gathered from it, and characterizing the application workload by

computing the proportion of requests handled by each path.

- We demonstrate the effectiveness of the approach using a working PaaS prototype.

5.1 Performance Debugging Cloud Applications

By providing most of the functionality that applications require via kernel services, the PaaS model significantly increases programmer productivity. However, a downside of this approach is that these features also hide the performance details of PaaS applications. Since the applications spend most of their time executing kernel services [134], it is challenging for the developers to debug performance issues given the opacity of the cloud platform's internal implementation.

One way to circumvent this problem is to instrument application code [12, 14, 13], and continuously monitor the time taken by various parts of the application. But such application-level instrumentation is tedious, and error prone thereby misleading those attempting to diagnose problems. Moreover, the instrumentation code may slow down or alter the application's performance. In contrast, implementing data collection and analysis as a kernel service built into the PaaS cloud allows performance diagnosis to be a "curated" service that is reliably managed by the cloud platform.

In order to maintain a satisfactory level of user experience and adhere to any previously agreed upon performance SLOs, application developers and cloud administrators wish to detect performance anomalies as soon as they occur. When detected, they must perform root cause analysis to identify the cause of the anomaly, and take some corrective and/or preventive action. This diagnosis usually occurs as a two step process. First, one must determine whether the anomaly was caused by a change in the workload (e.g. a sudden increase in the number of client requests). If that is the case, the resolution typically involves allocating more resources to the application or spawning more instances of the application for load balancing purposes. If the anomaly cannot be attributed to a workload change, one must go another step to find the bottleneck component that has given rise to the issue at hand.

5.2 Roots

Roots is a holistic system for application performance monitoring (APM), performance anomaly detection, and bottleneck identification. The key intuition behind the system is that, as an intrinsic PaaS service, Roots has visibility into all activities of the PaaS cloud, across layers. Moreover, since the PaaS applications we have observed spend most of their time in PaaS kernel services [134], we hypothesize that we can reason about application performance from observations of how the application uses the platform, i.e. by monitoring the time spent in PaaS kernel services. If we are able to do so, then we can avoid application instrumentation and its downsides while detecting performance anomalies, and identifying their root cause in near real time with low overhead.

The PaaS model that we assume with Roots is one in which the clients of a web application engage in a “service level agreement” (SLA) [89] with the “owner” or operator of the application that is hosted in a PaaS cloud. The SLA stipulates a response-time “service level objective” (SLO) that, if violated, constitutes a breech of the agreement. If the performance of an application deteriorates to the point that at least one of its SLOs is violated, we treat it as an *anomaly*. Moreover, we refer to the process of diagnosing the reason for an anomaly as *root cause analysis*. For a given anomaly, the root cause could be a change in the application workload or a *bottleneck* in the application runtime. Bottlenecks may occur in the application code, or in the PaaS kernel services that the application depends on.

Roots collects performance data across the cloud platform stack, and aggregates it based on request/response. It uses this data to infer application performance, and to identify SLO violations (performance anomalies). Roots can further handle different types of anomalies in different ways. We overview each of these functionalities in the remainder of this section.

5.2.1 Data Collection and Correlation

We must address two issues when designing a monitoring framework for a system as complex as a PaaS cloud.

1. Collecting data from multiple different layers.

2. Correlating data collected from different layers.

Each layer of the cloud platform is only able to collect data regarding the state changes that are local to it. A layer cannot monitor state changes in other layers due to the level of encapsulation provided by layers. However, processing an application request involves cooperation of multiple layers. To facilitate system-wide monitoring and bottleneck identification, we must gather data from all the different layers involved in processing a request. To combine the information across layers, we correlate the data, and link events related to the same client request together.

To enable this, we augment the front-end server of the cloud platform. Specifically, we have it tag incoming application requests with unique identifiers. This request identifier is added to the HTTP request as a header, which is visible to all internal components of the PaaS cloud. Next, we configure data collecting agents within the platform to record the request identifiers along with any events they capture. This way we record the relationship between application requests, and the resulting local state changes in different layers of the cloud, without breaking the existing level of abstraction in the

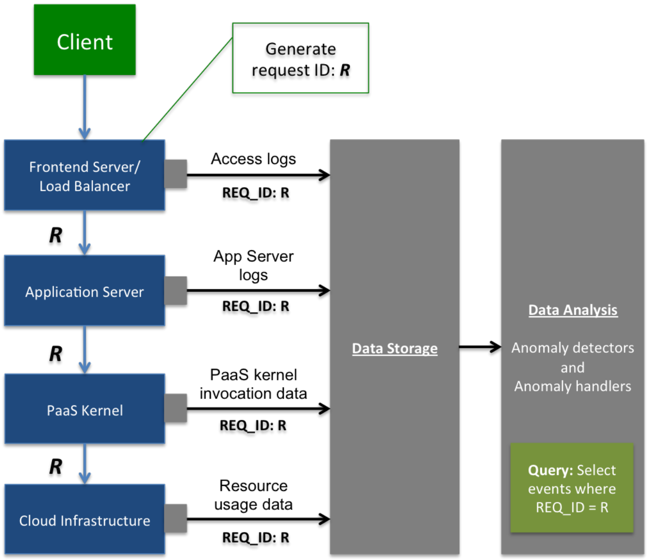

Figure 5.1: Roots APM architecture.

cloud architecture. This approach is also scalable, since the events are recorded in a distributed manner without having to maintain any state at the data collecting agents. Roots aggregates the recorded events by request identifier to efficiently group the related events as required during analysis.

Figure 5.1 illustrates the high-level architecture of Roots, and how it fits into the PaaS stack. APM components are shown in grey. The small grey boxes attached to the PaaS components represent the agents used to instrument the cloud platform. In the diagram, a user request is tagged with the identifier value R at the front-end server. This identifier is passed down to the lower layers of the cloud along with the request. Events that occur in the lower layers while processing this request are recorded with the request identifier R, so Roots can correlate them later. For example, in the data analysis component we can run a filter query to select all the events related to a particular request (as shown in the pseudo query in the diagram). Similarly, Roots can run a “group by” query to select all events, and aggregate them by the request identifier.

Figure 5.1 also depicts Roots data collection across all layers in the PaaS stack (i.e.

full stack monitoring). From the front-end server layer we gather information related to incoming application requests. This involves scraping the HTTP server access logs, which are readily available in most technologies used as front-end servers (e.g. Apache HTTPD, Nginx).

From the application server layer, we collect application logs and metrics from the application runtime that are easily accessible, e.g. process level metrics indicating resource usage of the individual application instances. Additionally, Roots employs a set of per-application benchmarking processes that periodically probes different applications to measure their performance. These are lightweight, stateless processes managed by the Roots framework. Data collected by these processes is sent to the data storage component, and is available for analysis as per-application time series data.

At the PaaS kernel layer we collect information regarding all kernel invocations made by the applications. This requires intercepting the PaaS kernel invocations at runtime. This must be done carefully so as to not introduce significant overhead application execution. For each PaaS kernel invocation, we capture the following parameters.

- Source application making the kernel invocation
- Timestamp
- A sequence number indicating the order of PaaS kernel invocations within an application request
- Target kernel service and operation
- Execution time of the invocation
- Request size, hash and other parameters

Collecting PaaS kernel invocation details enables tracing the execution of application requests without requiring that the application code be instrumented.

Finally, at the lowest level we can collect information related to virtual machines, containers and their resource usage. We gather metrics on network usage by individual components which is useful for traffic engineering use cases. We also scrape hypervisor and container manager logs to learn how resources are allocated and released over time.

To avoid introducing delays to the application request processing flow, we implement Roots data collecting agents as asynchronous tasks. That is, none of them suspend application request processing to report data to the data storage components. To enable this, we collect data into log files or memory buffers that are local to the components being monitored. This locally collected (or buffered) data is periodically sent to the data storage components of Roots using separate background tasks and batch communication operations. We also isolate the activities in the cloud platform from potential failures in the Roots data collection or storage components.

5.2.2 Data Storage

The Roots data storage is a database that supports persistently storing monitoring data, and running queries on them. Most data retrieval queries executed by Roots use application and time intervals as indices. Therefore a database that can index monitoring data by application and timestamp will greatly improve the query performance. It is also acceptable to remove old monitoring data to make room for more recent events, since Roots performs anomaly detection using the most recent data in near realtime.

5.2.3 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 deployed 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, which can be used to evaluate the efficiency of different detection strategies for any given application. 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 time range 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 encapsulates a unique anomaly identifier, timestamp, application identifier and the source detector's sliding window that correspond to the anomaly. Anomaly handlers are configured globally (i.e. each handler receives events from all detectors), but each handler can be programmed to handle only certain types of events. Furthermore, they can fire their own events, which are also delivered to all the listening anomaly handlers. Similar to detectors, Roots supports multiple anomaly handler implementations – one for logging anomalies, one for sending alert emails, one for updating a dashboard etc. Additionally, Roots provides two special anomaly handler implementations: a workload change analyzer, and a bottleneck identifier. We implement the communication between detectors and handlers using shared memory.

The ability of anomaly handlers to fire their own events, coupled with their support for responding to a filtered subset of incoming events enables constructing elaborate event flows with sophisticated logic. For example, the workload change analyzer can run some analysis upon receiving an anomaly event from any anomaly detector. If an anomaly cannot be associated with a workload change, it can fire a different type of event. The

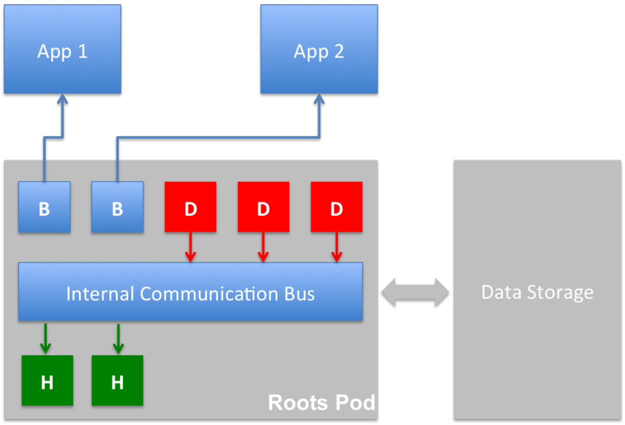

Figure 5.2: Anatomy of a Roots pod. The diagram shows 2 application benchmarking processes (B), 3 anomaly detectors (D), and 2 handlers (H). Processes communicate via a shared memory communication bus local to the pod.

bottleneck identifier, can be programmed to only execute its analysis upon receiving this second type of event. This way we perform the workload change analysis first, and perform the systemwide bottleneck identification only when it is necessary.

Both the anomaly detectors and anomaly handlers work with fixed-sized sliding windows. Therefore the amount of state these entities must keep in memory has a strict upper bound. The extensibility of Roots is primarily achieved through the abstractions of anomaly detectors and handlers. Roots makes it simple to implement new detectors and handlers, and plug them into the system. Both the detectors and the handlers are executed as lightweight processes that do not interfere with the rest of the processes in the cloud platform.

5.2.4 Roots Process Management

Most data collection activities in Roots can be treated as passive – i.e. they take place automatically as the applications receive and process requests in the cloud platform. They do not require explicit scheduling or management. In contrast, application benchmarking and data analysis are active processes that require explicit scheduling and management.

This is achieved by grouping benchmarking and data analysis processes into units called Roots pods.

Each Roots pod is responsible for starting and maintaining a preconfigured set of benchmarkers and data analysis processes (i.e. anomaly detectors and handlers). These processes are light enough, so as to pack a large number of them into a single pod. Pods are self-contained entities, and there is no inter-communication between pods. Processes in a pod can efficiently communicate with each other using shared memory, and call out to the central Roots data storage to retrieve collected performance data for analysis. This enables starting and stopping Roots pods with minimal impact on the overall monitoring system. Furthermore, pods can be replicated for high availability, and application load can be distributed among multiple pods for scalability.

Figure 5.2 illustrates a Roots pod monitoring two applications. It consists of two benchmarking processes, three anomaly detectors and two anomaly handlers. The anomaly detectors and handlers are shown communicating via an internal shared memory communication bus.

5.3 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 anomalies not caused by workload changes (HTTP request rate), Roots performs further analysis to identify the bottleneck component that is responsible for the issue.

We implement our prototype in AppScale [7], an open source PaaS cloud that is API compatible with Google App Engine (GAE) [4]. This compatibility enables us to evaluate our approach using real applications developed by others since GAE applications run on

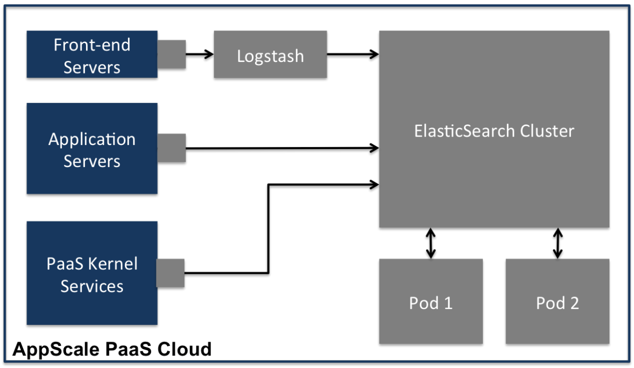

Figure 5.3: Roots prototype implementation for AppScale PaaS.

AppScale without modification. Because AppScale is open source, we were able to modify its implementation minimally to integrate Roots.

Figure 5.3 shows an overview of our prototype implementation. Roots components are shown in grey, while the PaaS components are shown in blue. We use ElasticSearch [135] as the data storage component of our prototype. ElasticSearch is ideal for storing large volumes of structured and semi-structured data [136]. ElasticSearch continuously organizes and indexes data, making the information available for fast and efficient querying. Additionally, it also provides powerful data filtering and aggregation features, which greatly simplify the implementations of high-level data analysis algorithms.

We configure AppScale's front-end server (based on Nginx) to tag all incoming application requests with a unique identifier. This identifier is attached to the incoming request as 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.

We implement a number of data collecting agents in AppScale to gather runtime information from all major components. These agents buffer data locally, and store them in ElasticSearch in batches. Events are buffered until the buffer accumulates 1MB of data, subject to a hard time limit of 15 seconds. 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. For scraping server logs, and storing the extracted entries in ElasticSearch, we use the Logstash tool [137]. To capture the PaaS kernel invocation data, we augment AppScale's PaaS kernel implementation, which is derived from the GAE PaaS SDK. More specifically we implement an agent that records all PaaS SDK calls, and reports them to ElasticSearch asynchronously.

We implement Roots pods as standalone Java server processes. Threads are used 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 itself. This way, our Roots implementation offloads heavy computations to ElasticSearch which is specifically designed for high-performance query processing and analytics. Some of the more sophisticated statistical analysis tasks (e.g. change point detection and linear regression as described below) are implemented in the R language, and the Roots pods integrate with R using the Rserve protocol [138].

5.3.1 SLO-violating Anomalies

As described previously, Roots defines anomalies as performance events that trigger SLO violations. Thus, we devise a detector to automatically identify when a SLO violation has occurred. This anomaly detector allows application developers to specify simple performance SLOs for deployed applications. A performance SLO consists of an upper bound on the application response time (T), and the probability (p) that the application response time falls under the specified upper bound. A general performance SLO can be stated as: "application responds under T milliseconds p% of the time".

When enabled for a given application, the SLO-based anomaly detector starts a benchmarking process that periodically measures the response time of the target application. Probes made by the benchmarking process are several seconds apart in time (sampling rate), so as to not strain the application with load. The detector then periodically analyzes the collected response time measurements to check if the application meets the specified performance SLO. Whenever it detects that the application has failed to meet the SLO, it triggers an anomaly event. The SLO-based anomaly detector supports following configuration parameters:

- Performance SLO: Response time upper bound (T), and the probability (p).
- Sampling rate: Rate at which the target application is benchmarked.
- Analysis rate: Rate at which the anomaly detector checks whether the application has failed to meet the SLO.
- Minimum samples: Minimum number of samples to collect before checking for SLO violations.
- Window size: Length of the sliding window (in time) to consider when checking for SLO violations. This imposes a limit on the number of samples to keep in memory.

Once the anomaly detector identifies an SLO violation, it will continue to detect the same violation until the historical data which contains the anomaly drops off from the sliding window. In order to prevent the detector from needlessly reporting the same anomaly multiple times, we purge all the data from anomaly detector's sliding window whenever it detects an SLO violation. Therefore, the detector cannot check for further SLO violations until it repopulates the sliding window with the minimum number of samples. This implies that each anomaly is followed by a "warm up" period. For instance, with a sampling rate of 15 seconds, and a minimum samples count of 100, the warm up period can last up to 25 minutes.

5.3.2 Path Distribution Analysis

We have implemented a path distribution analyzer in Roots whose function it is to identify recurring sequences of PaaS kernel invocations made by an application. Each identified sequence corresponds to a path of execution through the application code (i.e. a path through the control flow graph of the application). This detector is able to determine the frequency with which each path is executed over time. Then, using this information which we term a "path distribution," it reports an anomaly event when the distribution of execution paths changes.

For each application, a path distribution is comprised of the set of execution paths available in that application, along with the proportion of requests that executed each path. It is an indicator of the type of request workload handled by an application. For example, consider a data management application that has a read-only execution path, and a read-write execution path. If 90% of the requests execute the read-only path, and the remaining 10% of the requests execute the read-write path, we may characterize the request workload as read-heavy.

Roots path distribution analyzer facilitates computing the path distribution for each application with no static analysis, by only analyzing the runtime data gathered from the applications. It periodically computes the path distribution for a given application. If it detects that the latest path distribution is significantly different from the distributions seen in the past, it triggers an event. This is done by computing the mean request proportion for each path (over a sliding window of historical data), and then comparing the latest request proportion values against the means. If the latest proportion is off by more than n standard deviations from its mean, the detector considers it to be an anomaly. The sensitivity of the detector can be configured by changing the value of n, which defaults to 2.

Path distribution analyzer enables developers to know when the nature of their application request workload changes. For example in the previous data management application, if suddenly 90% of the requests start executing the read-write path, the Roots path distribution analyzer will detect the change. Similarly it is also able to detect when new paths of execution are being invoked by requests (a form of novelty detection).

5.3.3 Workload Change Analyzer

Performance anomalies can arise either due to bottlenecks in the cloud platform or changes in the application workload. When Roots detects a performance anomaly (i.e. an application failing to meet its performance SLO), it needs to be able to determine whether the failure is due to an increase in workload or a bottleneck that has suddenly manifested. To check if the workload of an application has changed recently, Roots uses a workload change analyzer. This Roots component is implemented as an anomaly handler, which gets executed every time an anomaly detector identifies a performance anomaly. Note that this is different from the path distribution analyzer, which is implemented as an anomaly detector. While the path distribution analyzer looks for changes in the *type* of the workload, the workload change analyzer looks for changes in the workload *size* or *rate*. In other words, it determines if the target application has received more requests than usual, which may have caused a performance degradation.

Workload change analyzer uses change point detection algorithms to analyze the historical trend of the application workload. We use the “number of requests per unit time” as the metric of workload size. Our implementation of Roots supports a number of well known change point detection algorithms (PELT [139], binary segmentation and CL method [140]), any of which can be used to detect level shifts in the workload size. Algorithms like PELT favor long lasting shifts (plateaus) in the workload trend, over momentary spikes. We expect momentary spikes to be fairly common in workload data. But it is the plateaus that cause request buffers to fill up, and consume server-side resources for extended periods of time, thus causing noticeable performance anomalies.

5.3.4 Bottleneck Identification

Applications running in the cloud consist of user code executed in the application server, and remote service calls to various PaaS kernel services. An AppScale cloud consists of the same kernel services present in the Google App Engine public cloud (datastore, memcache, urlfetch, blobstore, user management etc.). We consider each PaaS kernel invocation, and the code running on 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. The purpose of bottleneck identification is to find, out of all the components executed by an application, the one component that is most likely to have caused application performance to deteriorate.

Suppose an application makes n PaaS kernel invocations $(X_1, X_2, ...X_n)$ for each request. For any given application request, Roots captures the time spent on each kernel invocation $(T_{X_1}, T_{X_2}, ...T_{X_n})$, and the total response time (T_{total}) of the request. These time values are related by the formula $T_{total} = T_{X_1} + T_{X_2} + ... + T_{X_n} + r$, where r is all the time spent in the resident application server executing user code (i.e. the time spent not executing PaaS kernel services). r is not directly measured in Roots, since that requires code instrumentation. However, in previous work [134] we showed that typical PaaS-hosted web applications spend most of their time invoking PaaS kernel services. We make use of these findings, and assert that for typical, well-designed PaaS applications $r \ll T_{X_1} + T_{X_2} + ... + T_{X_n}$.

Roots bottleneck identification mechanism first selects up to four components as possible candidates for the bottleneck. These candidates are then further evaluated by a weighting algorithm to determine the actual bottleneck in the cloud platform.

Relative Importance of PaaS Kernel Invocations

The purpose of this metric is to find the component that is contributing the most towards the variance in the total response time. We select a window W in time which includes a sufficient number of application requests, and ending at the point when the performance anomaly was detected. Note that for each application request in W, we can fetch the total response time (T_{total}), and the time spent on individual PaaS kernel services (T_{X_n}) from the Roots data storage. We take all these T_{total} values and the corresponding T_{X_n} values in W, and fit a linear model of the form $T_{total} = T_{X_1} + T_{X_2} + ... + T_{X_n}$ using linear regression. Here we leave r out deliberately, since it is typically and ideally small.

Occasionally in AppScale, we observe a request where r is large relative to T_{X_n}. Often these rare events are correlated with large T_{X_n} values as well leading us to suspect that the effect may be due to an issue with the AppScale infrastructure (e.g. a major garbage collection event in the PaaS software). Overall, Roots detects these events, and identifies them correctly (as explained below), but they perturb the linear regression model. To prevent that, we filter out requests where the r value is too high. This is done by computing the mean (μ_r) and standard deviation (σ_r) of r over the selected window, and removing any requests where $r > \mu_r + 1.65\sigma_r$.

Once the regression model has been computed, we run a relative importance algorithm [141] to rank each of the regressors (i.e. T_{X_n} values) based on their contribution to the variance of T_{total}. We use the LMG method [142] which is resistant to multicollinearity, and provides a break down of the R^2 value of the regression according to how strongly each regressor influences the variance of the dependent variable. The relative importance values of the regressors add up to the R^2 of the linear regression. We consider $1 - R^2$

(the portion of variance in T_{total} not explained by the PaaS kernel invocations) as the relative importance of r. The component associated with the highest ranked regressor (i.e. highest relative importance) is chosen as a bottleneck candidate. Statistically, this is the component that causes the application response time to vary the most.

Changes in Relative Importance

Next we divide the time window W into equal-sized segments, and compute the relative importance metrics for regressors within each segment. We also compute the relative importance of r within each segment. This way we obtain a time series of relative importance values for each regressor and r. These time series represent how the relative importance of each component has changed over time.

We subject each relative importance time series to change point analysis to detect if the relative importance of any particular variable has increased recently. If such a variable can be found, then the component associated with that variable is also a potential candidate for the bottleneck. The candidate selected by this method represents a component whose performance has been stable in the past, and has become variable recently.

High Quantiles

Next we analyze the individual distributions of T_{X_n} and r. Recall that for each PaaS kernel invocation X_k, we have a distribution of T_{X_k} values in the window W. Similarly we can also extract a distribution of r values from W. Out of all the available distributions we 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. This component can be considered having a high latency in general.

Tail End Values

Finally, Roots analyzes each T_{X_k} and r distribution to identify the one with the largest tail values with respect to a particular high quantile. For each maximum (tail end) 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.

Selecting Among the 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.

5.4 Results

We evaluate the efficacy of Roots as a performance monitoring and root cause analysis system for PaaS 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. We also evaluate the Roots path distribution analyzer, and its ability to identify execution paths along with changes in path distributions. Finally, we investigate the performance and scalability of the Roots prototype.

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

Table 5.1: 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.

5.4.1 Anomaly Detection: Accuracy and Speed

To begin the evaluation of the Roots prototype we experiment with the SLO-based anomaly detector, 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 save the posted comments, and the AppScale user management service to handle authentication. Each request processed by guestbook results in two PaaS kernel 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 CPU cores (clocked at 2.4GHz), and 4GB of memory.

We run the SLO-based anomaly detector on guestbook with a sampling rate of 15 seconds, an analysis rate of 60 seconds, and a window size of 1 hour. We set the minimum sample 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 apriori 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. We denote the resulting three SLOs L_1, L_2 and L_3 respectively.

We also 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 guestbook deployment under test. Therefore this delay is sufficient to violate all three SLOs used in our experiments. We run a similar set of experiments where we inject faults into the user management service of AppScale. Each experiment is run for a period of 10 hours.

Table 5.1 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 detected some naturally occurring faults (temporary spikes in application latency), when a number of injected faults were 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.

Next we analyze how fast Roots can detect anomalies in an application. We first consider the performance of guestbook under the L_1 SLO while injecting faults into the datastore service. Figure 5.4 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, except in case of the fault injection window at 20:00 hours. Roots detected a 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

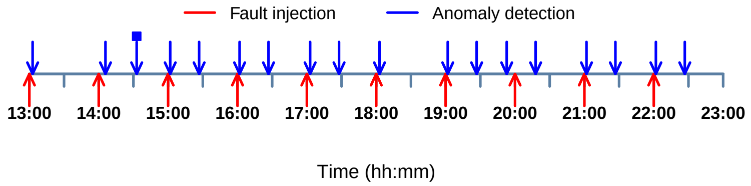

Figure 5.4: Anomaly detection in guestbook application during a period of 10 hours. Red arrows indicate fault injection at the datastore service. Blue arrows indicate all anomalies detected by Roots during the experimental run.

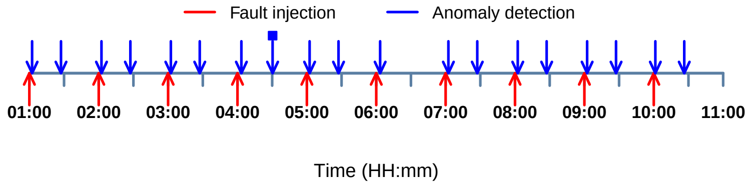

Figure 5.5: Anomaly detection in guestbook application during a period of 10 hours. Red arrows indicate fault injection at the user management service. Blue arrows indicate all anomalies detected by Roots during the experimental run.

20:00 hours. But as soon as the detector became active again at 20:17, it detected the anomaly.

Figure 5.5 shows the anomaly detection time line for the same application and SLO, while faults are being injected into the user management service. Here too we see that Roots detects anomalies immediately following each fault injection window.

5.4.2 Path Distribution Analyzer: Accuracy and Speed

Next we evaluate the effectiveness and accuracy of the path distribution analyzer. For this we employ two different applications.

key-value store This application provides the functionality of an online key-value store. It allows users to store data objects in the cloud where each object is assigned a unique key. The objects can then be retrieved, updated or deleted using their keys. Different operations (create, retrieve, update and delete) are implemented as

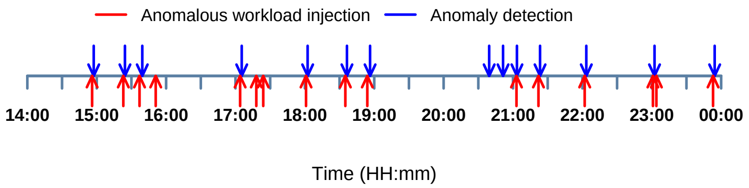

Figure 5.6: Anomaly detection in key-value store application during a period of 10 hours. Steady-state traffic is read-heavy. Red arrows indicate injection of write-heavy bursts. Blue arrows indicate all the anomalies detected by the path distribution analyzer.

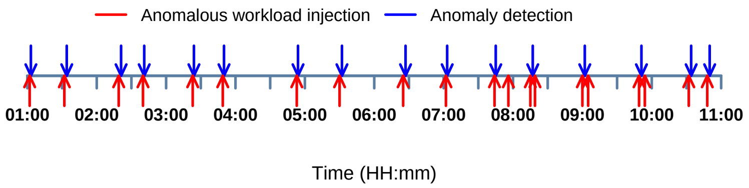

Figure 5.7: Anomaly detection in cached key-value store application during a period of 10 hours. Steady-state traffic is mostly served from the cache. Red arrows indicate injection of cache-miss bursts. Blue arrows indicate all the anomalies detected by the path distribution analyzer.

separate paths of execution in the application.

cached key-value store This is a simple extension of the regular key-value store, which adds caching to the read operation using the AppScale's memcache service. The application contains separate paths of execution for cache hits and cache misses.

We first deploy the key-value store on AppScale, and populate it with a number of data objects. Then we run a test client against it which generates a read-heavy workload. On average this workload consists of 90% read requests and 10% write requests. The test client is also programmed to randomly send bursts of write-heavy workloads. These bursts consist of 90% write requests on average, and each burst lasts up to 2 minutes. Figure 5.6 shows the write-heavy bursts as events on a time line (indicated by red arrows). Note that almost every burst is immediately followed by an anomaly detection event (indicated by blue arrows). The only time we do not see an anomaly detection event is when multiple bursts are clustered together in time (e.g. 3 bursts between 17:04 and 17:24 hours). In this case Roots detects the very first burst, and then goes into the warm up mode to collect more data. Between 20:30 and 21:00 hours we also had two instances where the read request proportion dropped from 90% to 80% due to random chance. This is because our test client randomizes the read request proportion around the 90% mark. Roots identified these two incidents also as anomalous.

We conduct a similar experiment using the cached key-value store. Here, we run a test client that generates a workload that is mostly served from the cache. This is done by repeatedly executing read requests on a small selected set of object keys. However, the client randomly sends bursts of traffic requesting keys that are not likely to be in the application cache, thus resulting in many cache misses. Each burst lasts up to 2 minutes. As shown in figure 5.7, Roots path distribution analyzer correctly detects the change in the workload (from many cache hits to many cache misses), nearly every time the test client injects a burst of traffic that triggers the cache miss path of the application. The only exception is when multiple bursts are clumped together, in which case only the first raises an alarm in Roots.

5.4.3 Workload Change Analyzer Accuracy

Next we evaluate the Roots workload change analyzer. In this experiment we run a varying workload against the key-value store application for 10 hours. 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 5.8 shows how the application workload has changed over time. The workload

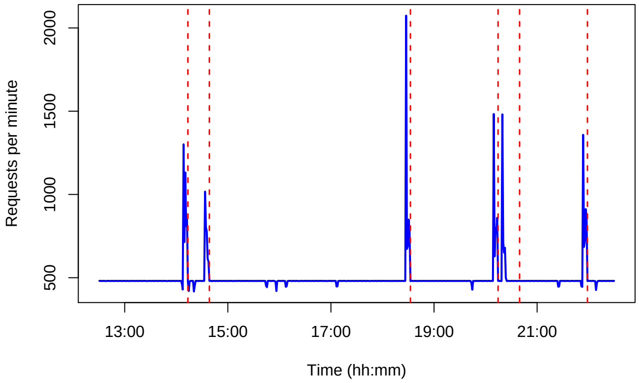

Figure 5.8: Workload size over time for the key-value store application. The test client randomly sends large bursts of traffic causing the spikes in the plot. Roots anomaly detection events are shown in red dashed lines.

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 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.

5.4.4 Bottleneck Identification Accuracy

Next 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 5.4, 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

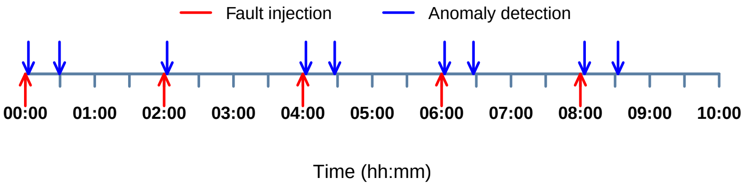

Figure 5.9: Anomaly detection in stock-trader application during a period of 10 hours. Red arrows indicate fault injection at the 1st datastore query. Blue arrows indicate all anomalies detected by Roots during the experimental run.

the datastore service is the one that was detected at 14:32 hours. This is indicated by the blue arrow with a small 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.

Similarly, in the experiment shown in figure 5.5, Roots determined that all the anomalies are caused by the user management service, except in one instance. This is again inline with our expectations since in this experiment we inject faults into the user management service. For the anomaly detected at 04:30 hours, Roots determined that local execution time is the primary bottleneck. Like earlier, we have manually verified this diagnosis to be accurate. In this case too the server hosting the guestbook application became slow during the 04:23 - 04:25 time window, and Roots correctly identified the bottleneck as the local application server.

In order to evaluate how the bottleneck identification performs when an application makes more than 2 PaaS kernel invocations, we conduct another experiment using an

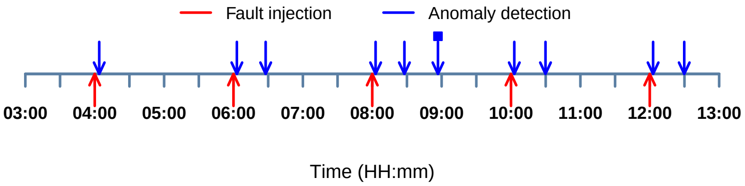

Figure 5.10: Anomaly detection in stock-trader application during a period of 10 hours. Red arrows indicate fault injection at the 2nd datastore query. Blue arrows indicate all anomalies detected by Roots during the experimental run.

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 *buy* and *sell*. Each of these operations makes 8 calls to the AppScale datastore. According to our previous work [134], 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. We execute a number of experimental runs using this application, and here we present the results from two of them. In all experiments 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 5.9 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 when a large number of injected faults were still 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 5.10 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 diagnosis 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.

In the experiments illustrated in figures 5.4, 5.5, 5.9, and 5.10 we maintain the application request rate steady throughout the 10 hour periods. Therefore, the workload change analyzer of Roots did not detect any significant shifts in the workload level. Consequently, none of the anomalies detected in these 4 experiments were attributed to a workload change. The bottleneck identification was therefore triggered for each anomaly.

To evaluate the agreement level among the four bottleneck candidate selection methods, we analyze 407 anomalies detected by Roots over a period of 3 weeks. We report that except on 13 instances, in all the remaining cases 2 or more candidate selection methods agreed on the final bottleneck component chosen. This implies that most of the time (96.8%) Roots identifies bottlenecks with high confidence.

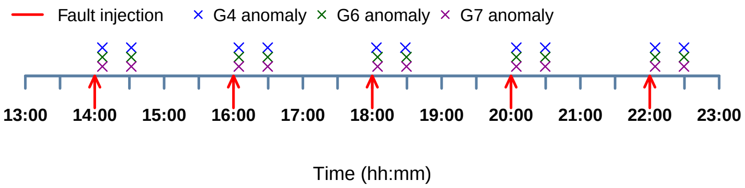

Figure 5.11: Anomaly detection in 8 applications deployed in a clustered AppScale cloud. Red arrows indicate fault injection at the datastore service for queries generated from a specific host. Cross marks indicate all the anomalies detected by Roots during the experiment.

5.4.5 Multiple Applications in a Clustered Setting

To demonstrate how Roots can be used in a multi-node environment, we set up an AppScale cloud on 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 multitenant support in AppScale to register each instance of guestbook as a different application (named $G1$ through $G8$). Each instance is hosted on a separate application server instance, has its own private namespace on the AppScale datastore, and can be accessed via a unique URL. We disable auto-scaling support in the AppScale cloud, and inject faults into the datastore service of AppScale in such a way that queries issued from a particular VM, are processed with a 100ms delay. We identify this 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, and when activated it lasts for up to 5 minutes. Then we 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 5.11 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 cloud revealed that these were the only 3 applications that were hosted on V_f. As a result, the bi-hourly fault injection caused their SLOs to get violated. Other applications did not exhibit any SLO violations since we are monitoring against a very high response time upper bound.

In each case Roots detected the SLO violations 2-3 minutes into the fault injection period. As soon as that happened, the anomaly detectors of $G4$, $G6$ and $G7$ entered the warmup mode. But our fault injection logic kept injecting faults for at least 2 more minutes. Therefore when the anomaly detectors reactivated after 25 minutes (time to collect the minimum sample count), they each detected another SLO violation. As a result, we see another set of detection events approximately half an hour after the fault injection events.

5.4.6 Results Summary

We conclude our discussion of Roots efficacy with a summary of our results. Table 5.2 provides an overview of all the results presented so far, broken down into four features that we wish to see in an anomaly detection and bottleneck identification system.

5.4.7 Roots Performance and Scalability

Next 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

Feature	Results Observed in Roots
Detecting anomalies	All the artificially induced anomalies were detected, except when multiple anomalies are clustered together in time. In that case only the first anomaly was detected. Roots also detected several anomalies that occurred due to a combination of injected faults, and natural faults.
Characterizing anomalies as being due to workload changes or bottlenecks	When anomalies were induced by varying the application workload, Roots correctly determined that the anomalies were caused by workload changes. In all other cases we kept the workload steady, and hence the anomalies were attributed to a system bottleneck.
Identifying correct bottleneck	In all the cases where bottleneck identification was performed, Roots correctly identified the bottleneck component.
Reaction time	All the artificially induced anomalies (SLO violations) were detected as soon as enough samples of the fault were taken by the benchmarking process (2-5 minutes from the start of the fault injection period).
Path distribution	All the artificially induced changes to the path distribution were detected.

Table 5.2: Summary of Roots efficacy results.

	Without Roots		With Roots	
App./Concurrency	Mean (ms)	SD	Mean (ms)	SD
guestbook/1	12	3.9	12	3.7
guestbook/50	375	51.4	374	53
stock-trader/1	151	13	145	13.7
stock-trader/50	3631	690.8	3552	667.7
kv store/1	7	1.5	8	2.2
kv store/50	169	26.7	150	25.4
cached kv store/1	3	2.8	2	3.3
cached kv store/50	101	24.8	97	35.1

Table 5.3: Latency comparison of applications when running on a vanilla AppScale cloud vs when running on a Roots-enabled AppScale cloud.

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 5.3 shows the comparison of request latencies. We discover 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. The request latency increases when the number of concurrent clients is increased from 1 to 50 (since all requests are handled by a single application server), but still there is no sign of any detrimental overhead from Roots even under load.

Finally, to demonstrate how lightweight and scalable Roots is, we deploy a Roots pod on a virtual machine with 4 CPU cores and 4GB memory. To simulate monitoring multiple applications, we run multiple concurrent anomaly detectors in the pod. Each

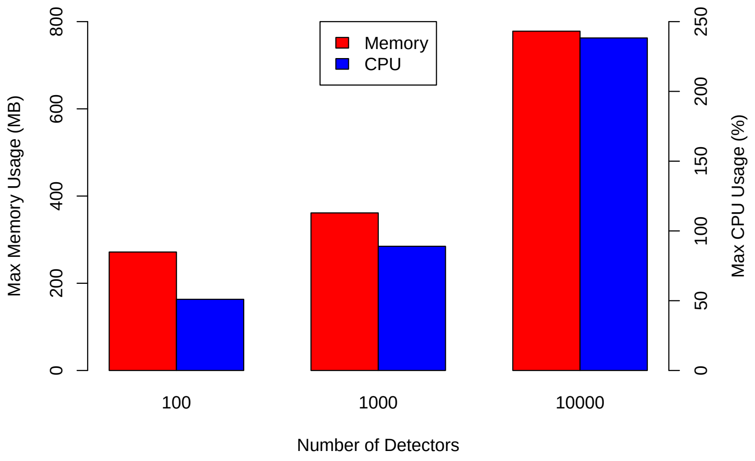

Figure 5.12: Resource utilization of a Roots pod.

detector is configured with a 1 hour sliding window. We vary the number of concurrent detectors between 100 and 10000, and run each configuration for 2 hours. We track the memory and CPU usage of the pod during each of these runs using the jstat and pidstat tools.

Figure 5.12 illustrates the maximum resource utilization of the Roots pod for different counts of concurrent anomaly detectors. We see that with 10000 concurrent detectors, the maximum CPU usage is 238%, where 400% is the available limit for 4 CPU cores. The maximum memory usage in this case is only 778 MB. Since each anomaly detector operates with a fixed-sized window, and they bring additional data into memory only when required, the memory usage of the Roots pod generally stays low. We also experimented with larger concurrent detector counts, and we were able to pack up to 40000 detectors into the pod before getting constrained by the CPU capacity of our VM. This result implies that we can monitor tens of thousands of applications using a single pod, thereby scaling up to a very large number of applications using only a handful of pods.

5.5 Related Work

Roots falls into the category of performance anomaly detection and bottleneck identification (PADBI) systems. A PADBI system is an entity that observes, in real time, the performance behaviors of a running system or application, while collecting vital measurements at discrete time intervals to create baseline models of typical system behaviors [133]. Such systems play a crucial role in achieving guaranteed service reliability, performance and quality of service by detecting performance issues in a timely manner before they escalate into major outages or SLO/SLA violations [143]. PADBI systems are thoroughly researched, and well understood in the context of traditional standalone and network applications. Many system administrators are familiar with frameworks like Nagios [144], Open NMS [145] and Zabbix [146] which can be used to collect data from a wide range of applications and devices.

However, the paradigm of cloud computing, being relatively new, 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 [147], New Relic [12] and DataDog [14] 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 under layers of managed services, and the application code is executed in a sandboxed environment that is not always amenable to instrumentation. When code instrumentation is possible, it tends to be burdensome, error prone, and detrimental to the application's performance. Roots on the other hand is built into the fabric of the PaaS cloud giving it full visibility into all the activities that take place in the entire software stack, and it does not require application-level instrumentation.

Our work borrows heavily from the past literature [132, 133] that detail the key features of cloud APMs. Consequently, we strive to incorporate requirements like scalability, autonomy and dynamic resource management into our design. Ibidunmoye et al highlight the importance of multilevel bottleneck identification as an open research question [133]. This is the ability to identify bottlenecks from a set of top-level application service components, and further down through the virtualization layer to system resource bottlenecks. Our plan for Roots is highly in sync with this vision. We currently support identifying bottlenecks from a set of kernel services provided by the PaaS cloud. As a part of our future work, we plan to extend this support towards the virtualization layer and the physical resources of the cloud platform.

Cherkasova et al developed an online performance modeling technique to detect anomalies in traditional transaction processing systems [148]. They divide time into contiguous segments, such that within each segment the application workload (volume and type of transactions) and resource usage (CPU) can be fit to a linear regression model. Segments for which a model cannot be found, are considered anomalous. Then they remove anomalous segments from the history, and perform model reconciliation to differentiate between workload changes and application problems. While this method is powerful, it requires instrumenting application code to detect different external calls (e.g. database queries) executed by the application. Since the model uses different transaction types as parameters, some prior knowledge regarding the transactions also needs to be fed into the system. The algorithm is also very compute intensive, due to continuous segmentation and model fitting. In contrast, we use a very lightweight SLO monitoring method in Roots to detect performance anomalies, and only perform heavy computations to perform bottleneck identification.

Dean et al implemented PerfCompass [149], an anomaly detection and localization method for IaaS clouds. They instrument the VM operating system kernels to capture the system calls made by user applications. Anomalies are detected by looking for unusual increases in system call execution time. They group system calls into execution units (processes, threads etc), and analyze how many units are affected by any given anomaly. Based on this metric they conclude if the problem was caused by a workload change or an application level issue. We take a similar approach in Roots, in that we capture the PaaS kernel invocations made by user applications. We use application response time (latency) as an indicator of anomalies, and group PaaS kernel invocations into application requests to perform bottleneck identification.

Nguyen et al presented PAL, another anomaly detection and localization mechanism targeting distributed applications deployed on IaaS clouds [150]. Similar to Roots, they also use an SLO monitoring approach to detect application performance anomalies. When an anomaly is detected, they perform change point analysis on gathered resource usage data (CPU, memory and network) to identify the anomaly onset time. Having detected one or more anomaly onset events in different components of the distributed application, they sort the events by time to determine the propagation pattern of the anomaly.

Magalhaes and Silva have made significant contributions in the area of anomaly detection and root cause analysis in web applications [151, 152]. They compute the correlation between application workload and latency. If the level of correlation drops significantly, they consider it to be an anomaly. A similar correlation analysis between workload and other local system metrics (e.g. CPU and memory usage) is used to identify the system resource that is responsible for a given anomaly. They also use an aspect-oriented programming model in their target applications, which allows them to easily instrument application code, and gather metrics regarding various remote services (e.g. database) invoked by the application. This data is subjected to a series of simple linear regressions to perform root cause analysis. This approach assumes that remote services are independent of each other. However, in a cloud platform where kernel services are deployed in the same shared infrastructure, this assumption might not hold true. Therefore we improve on their methodology, and use multiple linear regression with relative importance to identify cloud platform bottlenecks. Relative importance is resistant to multicollinearity, and therefore does not require the independence assumption.

Anomaly detection is a general problem not restricted to performance analysis. Researchers have studied anomaly detection from many different points of view, and as a result many viable algorithms and solutions have emerged over time [153]. Prior work in performance anomaly detection and root cause analysis can be classified as statistical methods (e.g. [154, 155, 152, 150]) and machine learning methods (e.g. [156, 157, 158]). 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 a framework on top of which new anomaly detectors and anomaly handlers can be built.

5.6 Conclusions and Future Work

Uncovering performance bottlenecks in a timely manner, and resolving them urgently is a key requirement for implementing governance in cloud environments. Application developers and cloud administrators wish to detect performance anomalies in cloud applications, and perform root cause analysis to diagnose problems. However, the high level of abstraction provided by cloud platforms, coupled with their scale and complexity, makes performance diagnosis a daunting problem. This situation is particularly apparent in PaaS clouds, where the application runtime details are hidden beneath a layer of kernel services. The existing cloud monitoring solutions do not have the necessary penetrative power to monitor all the different layers of cloud platforms, and consequently, their diagnosis capabilities are severely limited.

We present Roots, a near real time monitoring framework for 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 having to instrument the application code in anyway. Roots captures runtime data from all the different layers involved in processing application requests. It can correlate events across different layers, and identify bottlenecks deep within the kernel services of the PaaS.

Roots monitors applications for SLO compliance, and detects anomalies via SLO violations. When Roots detects an anomaly, it analyzes workload data and other application runtime data to perform root cause analysis. Roots is able to determine whether a particular anomaly was caused by a change in the application workload, or due to a bottleneck in the cloud platform. To this end we also devise a bottleneck identification algorithm, that uses a combination of linear regression, quantile analysis and change point detection. We also present an analysis method by which Roots can identify different paths of execution in an application. Our method does not require static analysis, and we use it to detect changes in an application's workload characteristics.

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, pinpointing the exact PaaS kernel 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 very lightweight, and scales well to handle large populations of applications.

In our 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. With that we expect to extend the root cause analysis support of Roots so that it can not only pinpoint the bottlenecked application components, but also the low level hosts and system resources that constitute each bottleneck.

Chapter 6

Conclusion

Cloud computing delivers IT infrastructure resources, programming platforms, and software applications as shared utility services. Enterprises and developers increasingly deploy applications on cloud platforms due to their scalability, high availability and many other productivity enhancing features. Cloud-hosted applications depend on the core services provided by the cloud platform for compute, storage and network resources. In some cases they use the services provided by the cloud to implement most of the application functionality as well (e.g. PaaS-hosted applications). Cloud-hosted applications are typically accessed over the Internet, via the web APIs exposed by the applications.

As the applications hosted in cloud platforms continue to increase in number, the need for enforcing governance on them becomes accentuated. We define governance as the mechanism by which the acceptable operational parameters are specified and maintained for a cloud-hosted application. Governance enables specifying the acceptable development standards and runtime parameters (performance, availability, security requirements etc.) for cloud-hosted applications as policies. Such policies can then be enforced automatically at various stages of the application life-cycle. Governance also entails monitoring cloud-hosted applications to ensure that they operate at a certain level of quality, and taking corrective action when deviations are detected. Through the steps of specification, enforcement, monitoring and correction, governance facilitates resolving a number of prevalent issues in today's cloud platforms. These issues include lack of good software engineering practices (code reuse, dependency management, versioning etc), lack of performance SLOs for cloud-hosted applications, and lack of performance debugging support.

We explore the feasibility of efficiently enforcing governance on cloud-hosted applications, and evaluate the effectiveness of governance as a means of achieving administrative conformance, developer best practices and performance SLOs in the cloud. Considering the scale of today's cloud platforms in terms of the number of users and the applications, we strive to automate much of the governance tasks through automated analysis and diagnostics. To achieve efficiency, we put more emphasis on deployment-time policy enforcement, static analysis of performance bounds, and non-invasive passive monitoring of cloud platforms, thereby keeping the governance overhead to a minimum. We avoid run-time enforcement and invasive instrumentation of cloud applications as much as possible. We also focus on building governance systems that are deeply integrated with the cloud platforms themselves. This enables using the existing scalability and high availability features of the cloud to provide an efficient governance solution that can control all application events in a fine-grained manner. Furthermore, such integrated solutions relieve the users from having to maintain and pay for additional, external governance and monitoring solutions.

In order to explore the feasibility of implementing efficient, automated governance systems in cloud environments, and evaluate the efficacy of such systems, we follow a three-step research plan.

1. Design and implement a scalable, low-overhead policy enforcement system for cloud

platforms.

2. Design and implement a methodology for formulating performance SLOs for cloud-hosted applications.
3. Design and implement a scalable application performance monitoring system for detecting and diagnosing performance anomalies in cloud platforms.

We design and implement EAGER [54, 91] – a lightweight governance policy enforcement framework built into PaaS clouds. It supports defining policies using a simple syntax based on the popular Python programming language. EAGER promotes deployment-time policy enforcement, where policies are enforced on user applications (and APIs) every time an application is uploaded to the cloud. By carrying out policy validations at application deployment-time, and refusing to deploy applications that violate policies, we provide fail-fast semantics, which ensure that deployed applications are fully policy compliant. EAGER architecture also provides the necessary provisions for facilitating run-time policy enforcement (through an API gateway proxy) when necessary. This is required, since not all policy requirements are enforceable at deployment-time; e.g. a policy that prevents an application from making connections to a specific network address. Our experimental results show that EAGER validation and policy enforcement overhead is negligibly small, and it scales well to handle thousands of user applications and policies. Overall, we show that integrated governance for cloud-hosted applications is not only feasible, but also can be implemented with very little overhead and effort.

To facilitate formulating performance SLOs, we design and implement Cerebro [134] – a system that predicts bounds on the response time of web applications developed for PaaS clouds. Cerebro is able to analyze a given web application, and determine a bound on its response time without subjecting the application to any testing or runtime instrumentation. This is achieved by a mechanism that combines static analysis of application source code with runtime monitoring of the underlying cloud platform (PaaS SDK to be specific). Our approach is limited to interactive web applications developed using a PaaS SDK. We show that such applications have very few branches and loops, and they spend most of their execution time invoking PaaS SDK operations. These properties make the applications amenable to both static analysis, and statistical treatment of their performance limits.

Cerebro is fast, can be invoked at the deployment-time of an application, and does not require any human input or intervention. The bounds predicted by Cerebro can be used as statistical guarantees (with well defined correctness probabilities) to form performance SLOs. These SLOs in turns can be used in SLAs that are negotiated with the users of the web applications. Cerebro's SLO prediction capability, coupled with a policy enforcement framework such as EAGER, can facilitate specification and enforcement of performance-related policies for cloud-hosted applications. We implement Cerebro for Google App Engine public cloud and AppScale private cloud. Our experiments with real world PaaS applications show that Cerebro is able to determine accurate performance SLOs that closely reflect the actual response time of the applications. Furthermore, we show that Cerebro-predicted SLOs are not easily affected by the dynamic nature of the cloud platform, and they remain valid for long durations. More specifically, Cerebro predictions remain correct for more than 12 days on average [159].

Finally, we design and implement Roots – a performance anomaly detection and bottleneck identification system built into PaaS clouds. It collects data from all the different layers of the PaaS stack; from load balancers to low level PaaS kernel service implementations. However, it does so without instrumenting user code, and without introducing a significant overhead to the application request processing flow. Roots uses the metadata (request identifiers) injected by the load balancers to correlate the events observed in different layers, thereby enabling tracing of application requests through the PaaS stack. Roots is also extensible in the sense that any number of statistical analysis methods can be incorporated into Roots for performance anomaly detection and diagnosis. Furthermore, it facilitates configuring monitoring requirements at the granularity of user applications, which allows different applications to be monitored and analyzed differently.

Roots detects performance anomalies by monitoring applications for performance SLO violations. When an anomaly (i.e. an SLO violation) is detected, Roots determines if the anomaly was caused by a change in the application workload or by a performance bottleneck in one of the underlying PaaS kernel services. If the SLO violation was caused by a performance bottleneck in the cloud, Roots needs to be able to locate the exact PaaS kernel service in which the bottleneck manifested. To this end we present a root cause analysis method that uses a combination of linear regression, change point detection and quantile analysis. We show that our combined methodology makes accurate diagnoses nearly 100% of the time. Moreover, we also present a path distribution analyzer that can identify different paths of execution in an application, via the run-time data gathered from the cloud platform. We show that this mechanism is capable of detecting characteristic changes in application workload as a special type of anomalies.

Our results demonstrate that efficient and automated governance in cloud environments is not only feasible, but also highly effective. We did not have to implement a cloud platform from the scratch to implement the governance systems designed as a part of this work. Rather, we were able to implement the proposed governance systems for existing cloud platforms like Google App Engine and AppScale; often with minimal changes to the cloud platform software. Our policy enforcement and monitoring systems are integrated with the cloud platform (i.e. they operate from within the cloud platform), and hence preclude the cloud platform users from having to set up or implement their own external governance solutions that provide API management or application monitoring functionality. Our governance systems are also efficient, in the sense they do not add a significant overhead to the applications deployed in the cloud platform, and they scale well to handle a very large number of applications and governance policies.

Our research is aimed at providing increased levels of oversight, control and automation to cloud platforms. Therefore it has the potential to increase the value offered by the cloud platforms to the application developers and the application clients. More specifically, our research can greatly enhance the use of PaaS clouds. A lot of our work is directly applicable to popular PaaS clouds such as Google App Engine and AppScale, and the respective developer communities can greatly benefit from our findings.

Our research paves the way to making cloud platforms more dependable and maintainable for administrators, application developers and clients alike. It brings automated policy enforcement – a governance technique that has been successfully applied in classic SOA systems in the past – to modern cloud environments. Policy enforcement solves a variety of issues related to poor application coding practices, and lack of administrative control. We also enable stipulating performance SLOs for cloud-hosted applications, a feature that is not supported in existing cloud platforms to the best of our knowledge. Our research also supports full-stack monitoring of cloud platforms for detecting performance SLO violations, and determining the root causes of such violations. When taken together, our research addresses all three components of governance (specification, enforcement and monitoring) both efficiently and automatically, as cloud-native features. The systems we propose ensure that cloud-hosted applications always operate in a policy compliant state, and any performance anomalies are detected and diagnosed fast. In conclusion, our governance systems facilitate achieving developer best practices, administrative conformance and performance SLOs for cloud-hosted applications in ways that were not possible before.

Bibliography

[1] Q. Hassan, *Demystifying cloud computing*, *The Journal of Defense Software Engineering* (2011) 16–21.

[2] P. M. Mell and T. Grance, *Sp 800-145. the nist definition of cloud computing*, tech. rep., Gaithersburg, MD, United States, 2011.

[3] *Amazon Web Services home page*, 2015. `http://aws.amazon.com/` [Accessed March 2015].

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

[5] "Microsoft windows azure." "http://www.microsoft.com/windowsazure/" [Accessed March 2015].

[6] 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.

[7] C. Krintz, *The appscale cloud platform: Enabling portable, scalable web application deployment*, *Internet Computing, IEEE* **17** (March, 2013) 72–75.

[8] "OpenShift by RedHat." "https://www.openshift.com".

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

[10] 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.

[11] "Roundup of Cloud Computing Forecasts and Market Estimates 2015." `http://www.forbes.com/sites/louiscolumbus/2015/01/24/roundup-of-cloud-computing-forecasts-and-market-estimates-2015` [Accessed May 2016].

[12] "Application Performance Monitoring and Management - New Relic." http://www.newrelic.com [Accessed April 2016].

[13] "Application Performance Monitoring and Management - Dynatrace." http://www.dynatrace.com [Accessed April 2016].

[14] "Datadog - Cloud-scale Performance Monitoring." http://www.datadoghq.com [Accessed April 2016].

[15] Brown, Allen E and Grant, Gerald G, *Framing the frameworks: A review of IT governance research*, *Communications of the Association for Information Systems* **15** (2005), no. 1 38.

[16] "Gartner, Magic Quadrant for Integrated SOA Governance Technology Sets, 2007." https://www.gartner.com/doc/572713/magic-quadrant-integrated-soa-governance [Accessed April 2016].

[17] "SOA Governance." http://www.opengroup.org/soa/source-book/gov/gov.htm. [Online; accessed 14-October-2013].

[18] T. G. J. Schepers, M. E. Iacob, and P. A. T. Van Eck, *A Lifecycle Approach to SOA Governance*, in *Proceedings of the 2008 ACM Symposium on Applied Computing*, 2008.

[19] F. Hojaji and M. R. A. Shirazi, *A Comprehensive SOA Governance Framework Based on COBIT*, in *2010 6th World Congress on Services*, 2010.

[20] K. Y. Peng, S. C. Lui, and M. T. Chen, *A Study of Design and Implementation on SOA Governance: A Service Oriented Monitoring and Alarming Perspective*, in *Service-Oriented System Engineering, 2008. SOSE '08. IEEE International Symposium on*, 2008.

[21] "Amazon Elastic Compute Cloud (Amazon EC2)." http://aws.amazon.com/ec2/.

[22] "Google Compute Engine IaaS." https://cloud.google.com/compute/.

[23] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, *Eucalyptus : A technical report on an elastic utility computing archietcture linking your programs to useful systems*, in *UCSB Technical Report ID: 2008-10*, 2008.

[24] "Heroku Cloud Application Platform." http://www.heroku.com.

[25] "Amazon Elastic Beanstalk." https://aws.amazon.com/elasticbeanstalk/.

[26] “Salesforce - What is SaaS?.” `https://www.salesforce.com/saas/`.

[27] “Workday - Alternative to ERP for HR and Financial Management.” `http://www.workday.com/`.

[28] “GoToMeeting - Easy Online Conferencing.” `http://www.gotomeeting.com`.

[29] *Protocol buffers*, 2016. `https://developers.google.com/protocol-buffers` [Accessed Sep 2016].

[30] 2009. `http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it` [Accessed Sep 2016].

[31] SearchCloudComputing, 2015. `http://searchcloudcomputing.techtarget.com/feature/Experts-forecast-the-2015-cloud-computing-market` [Accessed March 2015].

[32] Forbes, 2016. `http://www.forbes.com/sites/louiscolumbus/2016/03/13/roundup-of-cloud-computing-forecasts-and-market-estimates-2016` [Accessed Sep 2016].

[33] “Microsoft windows azure.” "http://www.microsoft.com/windowsazure/”.

[34] G. Ataya, *Information security, risk governance and management frameworks: An overview of cobit 5*, in *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN ’13, (New York, NY, USA), pp. 3–5, ACM, 2013.

[35] 2007. `http://www.isaca.org/certification/cgeit-certified-in-the-governance-of-enterprise-it/pages/default.aspx` [Accessed Sep 2016].

[36] M. P. Papazoglou, *Service-oriented computing: concepts, characteristics and directions*, in *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, 2003.

[37] “What is SOA?.” `http://www.opengroup.org/soa/source-book/soa/soa.htm` [Accessed April 2016].

[38] M. N. Haines and M. A. Rothenberger, *How a service-oriented architecture may change the software development process*, *Commun. ACM* **53** (Aug., 2010) 135–140.

[39] C. Xian-Peng, L. Bi-Ying, and M. Rui-Fang, *An ITIL v3-Based Solution to SOA Governance*, in *Services Computing Conference (APSCC), 2012 IEEE Asia-Pacific*, 2012.

[40] F. Belqasmi, R. Glitho, and C. Fu, *Restful web services for service provisioning in next-generation networks: a survey*, *Communications Magazine, IEEE* **49** (December, 2011) 66–73.

[41] A. M. Gutierrez, J. A. Parejo, P. Fernandez, and A. Ruiz-Cortes, *WS-Governance Tooling: SOA Governance Policies Analysis and Authoring*, in *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*, 2011.

[42] T. Phan, J. Han, J. G. Schneider, T. Ebringer, and T. Rogers, *A Survey of Policy-Based Management Approaches for Service Oriented Systems*, in *19th Australian Conference on Software Engineering (aswec 2008)*, 2008.

[43] Y. C. Zhou, X. P. Liu, E. Kahan, X. N. Wang, L. Xue, and K. X. Zhou, *Context Aware Service Policy Orchestration*, in *IEEE International Conference on Web Services (ICWS 2007)*, 2007.

[44] R. Strum, W. Morris, and M. Jander, *Foundations of Service Level Management.* Pearson, 2000.

[45] "Free and Enterprise API Management Platform and Infrastructure by 3scale – `http://www.3scale.net`."

[46] "Enterprise API Management and API Strategy – `http://apigee.com/about/`."

[47] "Enterprise API Management - Layer 7 Technologies – `http://www.layer7tech.com`."

[48] "ProgrammableWeb." `http://www.programmableweb.com` [Accessed March 2015].

[49] "ProgrammableWeb Blog – `http://blog.programmableweb.com/2013/04/30/9000-apis-mobile-gets-serious/`."

[50] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000. AAI9980887.

[51] *IEEE Xplore Search Gateway*, 2015. `http://ieeexplore.ieee.org/gateway/` [Accessed March 2015].

[52] *Berkeley API Central*, 2015. `https://developer.berkeley.edu` [Accessed March 2015].

[53] *Agency Application Programming Interfaces*, 2015. `http://www.whitehouse.gov/digitalgov/apis` [Accessed March 2015].

[54] C. Krintz, H. Jayathilaka, S. Dimopoulos, A. Pucher, R. Wolski, and T. Bultan, *Cloud platform support for api governance*, in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, 2014.

[55] A. S. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez, and U. Yalcinalp, *Web services policy framework (wspolicy)*, September, 2007.

[56] "SOA Governance Technical Standard – `http://www.opengroup.org/soa/source-book/gov/intro.htm`."

[57] C. Krintz, *The AppScale Cloud Platform: Enabling Portable, Scalable Web Application Deployment*, *IEEE Internet Computing* **Mar/Apr** (2013).

[58] G. Lawton, *Developing software online with platform-as-a-service technology*, *Computer* **41** (June, 2008) 13–15.

[59] "Platform as a Service - Pivotal CF." "http://www.gopivotal.com/platform-as-a-service/pivotal-cf".

[60] H. Jayathilaka, C. Krintz, and R. Wolski, *Towards Automatically Estimating Porting Effort between Web Service APIs*, in *Services Computing (SCC), 2014 IEEE International Conference on*, 2014.

[61] "Web Application Description Language." `http://www.w3.org/Submission/wadl/`, 2013. [Online; accessed 27-September-2013].

[62] "Swagger: A simple, open standard for describing REST APIs with JSON." `https://developers.helloreverb.com/swagger/`. [Online; accessed 05-August-2013].

[63] C. A. R. Hoare, *An axiomatic basis for computer programming*, *Commun. ACM* **12** (Oct., 1969) 576–580.

[64] H. Jayathilaka, A. Pucher, C. Krintz, and R. Wolski, *Using syntactic and semantic similarity of Web APIs to estimate porting effort*, *International Journal of Services Computing* **2** (2014), no. 4.

[65] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Vallés, and R. Van de Walle, *Functional descriptions as the bridge between hypermedia APIs and the Semantic Web*, in *International Workshop on RESTful Design*, 2012.

[66] T. Steiner and J. Algermissen, *Fulfilling the hypermedia constraint via http options, the http vocabulary in rdf, and link headers*, in *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11, (New York, NY, USA), pp. 11–14, ACM, 2011.

[67] “OAuth 2.0 – `http://oauth.net/2/`.”

[68] “Apache Synapse.” `https://synapse.apache.org/`. [Online; accessed 25-March-2014].

[69] “JSR311 - The Java API for RESTful Web Services – `https://jcp.org/aboutJava/communityprocess/final/jsr311/`.”

[70] “Swagger - A simple, open standard for describing REST APIs with JSON – `https://helloreverb.com/developers/swagger`.”

[71] “WSO2 API Manager.” `http://wso2.com/products/api-manager/`, 2013. [Online; accessed 27-September-2013].

[72] “WSO2 API Manager – `http://wso2.com/products/api-manager/`.”

[73] H. Guan, B. Jin, J. Wei, W. Xu, and N. Chen, *A framework for application server based web services management*, in *Software Engineering Conference, 2005. APSEC ’05. 12th Asia-Pacific*, pp. 8 pp.–, Dec, 2005.

[74] J. Wu and Z. Wu, *Dart-man: a management platform for web services based on semantic web technologies*, in *Computer Supported Cooperative Work in Design, 2005. Proceedings of the Ninth International Conference on*, vol. 2, pp. 1199–1204 Vol. 2, May, 2005.

[75] X. Zhu and B. Wang, *Web service management based on hadoop*, in *Service Systems and Service Management (ICSSSM), 2011 8th International Conference on*, pp. 1–6, June, 2011.

[76] C.-F. Lin, R.-S. Wu, S.-M. Yuan, and C.-T. Tsai, *A web services status monitoring technology for distributed system management in the cloud*, in *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2010 International Conference on*, pp. 502–505, Oct, 2010.

[77] S. Kikuchi and T. Aoki, *Evaluation of operational vulnerability in cloud service management using model checking*, in *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, pp. 37–48, March, 2013.

[78] Y. Sun, Z. Xiao, D. Bao, and J. Zhao, *An architecture model of management and monitoring on cloud services resources*, in *Advanced Computer Theory and Engineering (ICACTE)*, vol. 3, pp. V3–207–V3–211, Aug, 2010.

[79] R. Bhatti, D. Sanz, E. Bertino, and A. Ghafoor, *A policy-based authorization framework for web services: Integrating xgtrbac and ws-policy*, in *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pp. 447–454, July, 2007.

[80] S.-C. Chou and J.-Y. Jhu, *Access control policy embedded composition algorithm for web services*, in *Advanced Information Management and Service (IMS), 2010 6th International Conference on*, pp. 54–59, Nov, 2010.

[81] L. Li, K. Xiaohui, L. Yuanling, X. Fei, Z. Tao, and C. YiMin, *Policy-based fault diagnosis technology for web service*, in *Instrumentation, Measurement, Computer, Communication and Control, 2011 First International Conference on*, pp. 827–831, Oct, 2011.

[82] H. Liang, W. Sun, X. Zhang, and Z. Jiang, *A policy framework for collaborative web service customization*, in *Service-Oriented System Engineering, 2006. SOSE '06. Second IEEE International Workshop*, pp. 197–204, Oct, 2006.

[83] A. Erradi, P. Maheshwari, and S. Padmanabhuni, *Towards a policy-driven framework for adaptive web services composition*, in *Next Generation Web Services Practices, 2005. NWeSP 2005. International Conference on*, pp. 6 pp.–, Aug, 2005.

[84] A. Erradi, P. Maheshwari, and V. Tosic, *Policy-driven middleware for self-adaptation of web services compositions*, in *International Conference on Middleware*, 2006.

[85] B. Suleiman and V. Tosic, *Integration of uml modeling and policy-driven management of web service systems*, in *ICSE Workshop on Principles of Engineering Service Oriented Systems*, 2009.

[86] M. Thirumaran, D. Ponnurangam, K. Rajakumari, and G. Nandhini, *Evaluation model for web service change management based on business policy enforcement*, in *Cloud and Services Computing (ISCOS), 2012 International Symposium on*, pp. 63–69, Dec, 2012.

[87] F. Zhang, J. Gao, and B.-S. Liao, *Policy-driven model for autonomic management of web services using mas*, in *Machine Learning and Cybernetics, 2006 International Conference on*, pp. 34–39, Aug, 2006.

[88] "Mashery – `http://www.mashery.com`."

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

[90] D. Nurmi, J. Brevik, and R. Wolski, *QBETS: Queue Bounds Estimation from Time Series*, in *International Conference on Job Scheduling Strategies for Parallel Processing*, 2008.

[91] H. Jayathilaka, C. Krintz, and R. Wolski, *EAGER: Deployment-time API Governance for Modern PaaS Clouds*, in *IC2E Workshop on the Future of PaaS*, 2015.

[92] *Google App Engine Java Sandbox*, 2015. "https://cloud.google.com/appengine/docs/java/#Java_ The_sandbox" [Accessed March 2015].

[93] "Microsoft Azure Cloud SDK Service Quotas and Limits." `http://azure.microsoft.com/en-us/documentation/articles/azure-subscription-service-limits/#cloud-service-limits` [Accessed March 2015].

[94] "Google Cloud SDK Service Quotas and Limits." `https://cloud.google.com/appengine/docs/quotas` [Accessed March 2015].

[95] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, *Soot: A Java Bytecode Optimization Framework*, in *CASCON First Decade High Impact Papers*, 2010.

[96] *Github - build software better, together*, 2015. "https://github.com" [Accessed March 2015].

[97] F. E. Allen, *Control Flow Analysis*, in *Symposium on Compiler Optimization*, 1970.

[98] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[99] R. Morgan, *Building an Optimizing Compiler.* Digital Press, Newton, MA, USA, 1998.

[100] S. S. Muchnick, *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[101] S. Bygde, *Static WCET analysis based on abstract interpretation and counting of elements.* PhD thesis, Mälardalen University, 2010.

[102] `https://cloud.google.com/appengine/docs/java/javadoc/com/google/appengine/api/datastore/FetchOptions` [Accessed March 2015].

[103] D. Nurmi, J. Brevik, and R. Wolski, *Modeling Machine Availability in Enterprise and Wide-area Distributed Computing Environments*, in *Proceedings of Europar 2005*, 2005.

[104] J. Brevik, D. Nurmi, and R. Wolski, *Quantifying Machine Availability in Networked and Desktop Grid Systems*, in *Proceedings of CCGrid04*, April, 2004.

[105] R. Wolski and J. Brevik, *QPRED: Using Quantile Predictions to Improve Power Usage for Private Clouds*, Tech. Rep. UCSB-CS-2014-06, Computer Science Department of the University of California, Santa Barbara, Santa Barbara, CA 93106, September, 2014.

[106] D. Nurmi, R. Wolski, and J. Brevik, *Model-Based Checkpoint Scheduling for Volatile Resource Environments*, in *Proceedings of Cluster 2005*, 2004.

[107] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, *The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools*, *ACM Trans. Embed. Comput. Syst.* **7** (May, 2008).

[108] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper, *Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis.*, in *WCET*, 2007.

[109] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper, *Faster WCET Flow Analysis by Program Slicing*, in *ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, 2006.

[110] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen, *WCET Analysis of Java Bytecode Featuring Common Execution Environments*, in *International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2011.

[111] P. Cousot and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977.

[112] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, *A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models*, in *IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.

[113] S. Gulwani, S. Jain, and E. Koskinen, *Control-flow Refinement and Progress Invariants for Bound Analysis*, in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[114] S. Gulwani, K. K. Mehra, and T. Chilimbi, *SPEED: Precise and Efficient Static Estimation of Program Computational Complexity*, in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[115] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, *Comprehensive QoS Monitoring of Web Services and Event-based SLA Violation Detection*, in *International Workshop on Middleware for Service Oriented Computing*, 2009.

[116] A. K. Tripathy and M. R. Patra, *Modeling and Monitoring SLA for Service Based Systems*, in *International Conference on Intelligent Semantic Web-Services and Applications*, 2011.

[117] F. Raimondi, J. Skene, and W. Emmerich, *Efficient Online Monitoring of Web-service SLAs*, in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

[118] A. Bertolino, G. De Angelis, A. Sabetta, and S. Elbaum, *Scaling Up SLA Monitoring in Pervasive Environments*, in *Workshop on Engineering of Software Services for Pervasive Environments*, 2007.

[119] K. Mahbub and G. Spanoudakis, *Proactive SLA Negotiation for Service Based Systems: Initial Implementation and Evaluation Experience*, in *IEEE International Conference on Services Computing*, 2011.

[120] E. Yaqub, R. Yahyapour, P. Wieder, C. Kotsokalis, K. Lu, and A. I. Jehangiri, *Optimal negotiation of service level agreements for cloud-based services through autonomous agents*, in *IEEE International Conference on Services Computing*, 2014.

[121] L. Wu, S. Garg, R. Buyya, C. Chen, and S. Versteeg, *Automated SLA Negotiation Framework for Cloud Computing*, in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2013.

[122] T. Chau, V. Muthusamy, H.-A. Jacobsen, E. Litani, A. Chan, and P. Coulthard, *Automating SLA Modeling*, in *Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, 2008.

[123] K. Stamou, V. Kantere, J.-H. Morin, and M. Georgiou, *A SLA Graph Model for Data Services*, in *International Workshop on Cloud Data Management*, 2013.

[124] J. Skene, D. D. Lamanna, and W. Emmerich, *Precise Service Level Agreements*, in *International Conference on Software Engineering*, 2004.

[125] H. He, Z. Ma, H. Chen, and W. Shao, *Towards an SLA-Driven Cache Adjustment Approach for Applications on PaaS*, in *Asia-Pacific Symposium on Internetware*, 2013.

[126] C. Ardagna, E. Damiani, and K. Sagbo, *Early Assessment of Service Performance Based on Simulation*, in *IEEE International Conference on Services Computing (SCC)*, 2013.

[127] D. Dib, N. Parlavantzas, and C. Morin, *Meryn: Open, SLA-driven, Cloud Bursting PaaS*, in *Proceedings of the First ACM Workshop on Optimization Techniques for Resources Management in Clouds*, 2013.

[128] A. Iosup, N. Yigitbasi, and D. Epema, *On the Performance Variability of Production Cloud Services*, in *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, 2011.

[129] P. Leitner, B. Wetzstein, F. Rosenberg, A. Michlmayr, S. Dustdar, and F. Leymann, *Runtime Prediction of Service Level Agreement Violations for Composite Services*, in *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops* (A. Dan, F. Gittler, and F. Toumani, eds.), vol. 6275 of *Lecture Notes in Computer Science*, pp. 176–186. Springer Berlin Heidelberg, 2010.

[130] B. Tang and M. Tang, *Bayesian Model-Based Prediction of Service Level Agreement Violations for Cloud Services*, in *Theoretical Aspects of Software Engineering Conference (TASE)*, 2014.

[131] S. Duan and S. Babu, *Proactive Identification of Performance Problems*, in *ACM SIGMOD International Conference on Management of Data*, 2006.

[132] 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*, SAC '16, (New York, NY, USA), pp. 378–383, ACM, 2016.

[133] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, *Performance anomaly detection and bottleneck identification*, *ACM Comput. Surv.* **48** (2015), no. 1.

[134] 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.

[135] *Elasticsearch - search and analyze data in real time*, 2016. "https://www.elastic.co/products/elasticsearch" [Accessed Sep 2016].

[136] 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*, MSR 2014, (New York, NY, USA), pp. 328–331, ACM, 2014.

[137] *Logstash - collect, enrich and transport data*, 2016. "https://www.elastic.co/products/logstash" [Accessed Sep 2016].

[138] S. Urbanek, *Rserve – a fast way to provide r functionality to applications*, in *Proc. of the 3rd international workshop on Distributed Statistical Computing (DSC 2003)*, 2003.

[139] R. Killick, P. Fearnhead, and I. A. Eckley, *Optimal detection of changepoints with a linear computational cost*, *Journal of the American Statistical Association* **107** (2012), no. 500 1590–1598.

[140] C. Chen and L.-M. Liu, *Joint estimation of model parameters and outlier effects in time series*, *Journal of the American Statistical Association* **88** (1993), no. 421 284–297.

[141] U. Groemping, *Relative importance for linear regression in r: The package relaimpo*, *Journal of Statistical Software* **17** (2006), no. 1.

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

[143] 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*, pp. 83–90, Aug, 2011.

[144] R. C. Harlan, *Network management with nagios*, *Linux J.* **2003** (July, 2003) 3–.

[145] "The OpenNMS Project." `http://www.opennms.org` [Accessed April 2016].

[146] P. Tader, *Server monitoring with zabbix*, *Linux J.* **2010** (July, 2010).

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

[148] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, *Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change*, in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pp. 452–461, June, 2008.

[149] 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*, HotCloud'14, (Berkeley, CA, USA), pp. 16–16, USENIX Association, 2014.

[150] 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*, SLAML '11, (New York, NY, USA), pp. 1:1–1:8, ACM, 2011.

[151] 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*, NCA '10, (Washington, DC, USA), pp. 60–67, IEEE Computer Society, 2010.

[152] 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.

[153] V. Chandola, A. Banerjee, and V. Kumar, *Anomaly detection: A survey*, *ACM Comput. Surv.* **41** (July, 2009) 15:1–15:58.

[154] G. Casale, N. Mi, L. Cherkasova, and E. Smirni, *Dealing with burstiness in multi-tier applications: Models and their parameterization*, *IEEE Transactions on Software Engineering* **38** (Sept, 2012) 1040–1053.

[155] S. Malkowski, M. Hedwig, J. Parekh, C. Pu, and A. Sahai, *Bottleneck detection using statistical intervention analysis*, in *Proceedings of the Distributed Systems: Operations and Management 18th IFIP/IEEE International Conference on Managing Virtualization of Networks and Services*, DSOM'07, (Berlin, Heidelberg), pp. 122–134, Springer-Verlag, 2007.

[156] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, *Correlating instrumentation data to system states: A building block for automated diagnosis and control*, in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 16–16, USENIX Association, 2004.

[157] L. Yu and Z. Lan, *A scalable, non-parametric anomaly detection framework for hadoop*, in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, (New York, NY, USA), pp. 22:1–22:2, ACM, 2013.

[158] K. Bhaduri, K. Das, and B. L. Matthews, *Detecting abnormal machine characteristics in cloud infrastructures*, in *2011 IEEE 11th International Conference on Data Mining Workshops*, pp. 137–144, IEEE, 2011.

[159] H. Jayathilaka, C. Krintz, and R. Wolski, *Service-level agreement durability for web service response time*, in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.