Performance Problem Diagnosis in Cloud Infrastructures

Olumuyiwa Ibidunmoye

LICENTIATE THESIS, MAY 2016
DEPARTMENT OF COMPUTING SCIENCE
UMEÅ UNIVERSITY
SWEDEN

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

muyi@cs.umu.se

ISBN 978-91-7601-500-1
ISSN 0348-0542
UMINF 16.14

Printed by Print & Media, Umeå University, 2016

Abstract

Cloud datacenters comprise hundreds or thousands of disparate application services, each having stringent performance and availability requirements, sharing a finite set of heterogeneous hardware and software resources. The implication of such complex environment is that the occurrence of performance problems, such as slow application response and unplanned downtimes, has become a norm rather than exception resulting in decreased revenue, damaged reputation, and huge human-effort in diagnosis. Though causes can be as varied as application issues (e.g. bugs), machine-level failures (e.g. faulty server), and operator errors (e.g. mis-configurations), recent studies have attributed capacity-related issues, such as resource shortage and contention, as the cause of most performance problems on the Internet today. As cloud datacenters become increasingly autonomous there is need for automated performance diagnosis systems that can adapt their operation to reflect the changing workload and topology in the infrastructure. In particular, such systems should be able to detect anomalous performance events, uncover manifestations of capacity bottlenecks, localize actual root-cause(s), and possibly suggest or actuate corrections.

This thesis investigates approaches for diagnosing performance problems in cloud infrastructures. We present the outcome of an extensive survey of existing research contributions addressing performance diagnosis in diverse systems domains. We also present models and algorithms for detecting anomalies in real-time application performance and identification of anomalous datacenter resources based on operational metrics and spatial dependency across datacenter components. Empirical evaluations of our approaches shows how they can be used to improve end-user experience, service assurance and support root-cause analysis.

Preface

This thesis contains an introduction to performance management and diagnosis in cloud computing infrastructures, and the following papers.

Paper I Ibidunmoye, O., Hernández-Rodriguez, F., and Elmroth, E. Performance Anomaly Detection and Bottleneck Identification. *ACM Computing Surveys (CSUR): Volume 48, Issue 1, July 2015*.

Paper II Metsch, T., Ibidunmoye, O., Bayon-Molino, V., Butler, J., Hernández-Rodriguez, F., and Elmroth, E. Apex Lake: A Framework for Enabling Smart Orchestration. *Proceedings of the Industrial Track of the 16th International Middleware Conference*, December, 2015.

Paper III Ibidunmoye, O., Metsch, T., Bayon-Molino, V., and Elmroth, E. Performance Anomaly Detection using Datacenter Landscape Graphs. *To be submitted*, 2016.

Paper IV Ibidunmoye, O., Metsch, T., and Elmroth, E. Real-time Detection of Performance Anomalies for Cloud Services. *To be submitted*, 2016.

This work has been funded in part by the Swedish Research Council (VR) under grant agreement C0590801 (Cloud Control), the Swedish Strategic Research Program eSSENCE, and the European Union's Seventh Framework Programme under grant agreement 610711 (CACTOS).

Acknowledgments

To **Erik Elmroth**, my supervisor, I express my deepest gratitude for believing in me enough to accept me into his group and most importantly for mentoring, advising and guiding me through the on-going doctoral journey. I also appreciate my former assistant supervisor, **Francisco Hernandez Rodriguez**, for his guidance in the early periods of my study.

I'm really honoured to be part of the Distributed Systems group, how else could I ever have met such resourceful set of people. You guys have and are continuing to impact and challenge me in ways that are simply unquantifiable. I appreciate the senior colleagues (Erik, **Johan**, **P-O**, **Cristian**, and **Luis**) for the conducive environment; the freshly minted doctors (**Mina**, **Ahmed**, and **Ewnetu**) for constantly inspiring me, as well as colleagues still in the race (**Selome**, **Jakub**, **Gonzalo**, **Amardeep**, **Abel**, and **Chanh**) for the good friendship and support base. It's a privilege to have these professional software developers next door; **Lars** and **Peter**, you two have been helpful many a time. And to past members of the group, **Daniel**, **Petter** and **Viali**!, it's encouraging to see you guys do well in your careers and even more assuring to know that you're always there when we beckon. Again, special thanks to Cristian and Ewnetu for the many discussions, deliberation, tools, and guidance.

Many thanks to all staff of the IT-Support unit, especially **Tomas Forsman**, for always helping with technicalities and knotty IT stuff. I also appreciate the support of the Cloud Services Lab, Intel Labs Europe, Ireland; to **Thijs Metsch** I say thanks so much!

To God Almighty, I say a big 'Thank You!' for bringing me this far and for giving me the grace, courage and strength to push forward against all odds. I appreciate my parents and other members of my family for their moral support and good wishes. What could I do without my loving wife, my source of peace and courage in those dark times and in the face of weakness? **Wemimo**, I can't be grateful enough to you for delaying your career and comfort to stick up with me far away from home. Thanks for having my back always.

Umeå, May 2016
Olumuyiwa Ibidunmoye

Contents

Chapter 1

Introduction

The fundamental idea behind cloud computing was first conceived by John McCarthy as far back as 1961. He envisioned that "computation may someday be organized as a public utility" so as to reduce the cost of computing, increase reliability, and efficiency by relieving users from the need to own and operate complex computing infrastructure [1, 2]. Advancement in legacy technologies such as distributed systems, commodity computing, virtualization, and Grid computing has led to the realization of the cloud computing paradigm [3].

As the flexibility and scalability provided by cloud computing continues to improve the agility, responsiveness, and effectiveness of large-scale applications and organizations, meeting reliability, availability, and performance requirements remain a major source of concern. Studies [4, 3] have demonstrated through extensive experiments that performance variability is an important concern for cloud application providers because of its impact on end-user's quality of experience. In severe cases, unstable performance may manifest high-level symptoms such as degraded quality of service (e.g. slow page response) and service downtimes (e.g. unreachable service end-points). The reason for such performance problems is multi-faceted. Unhealthy resource interference due to time-sharing of heterogeneous application services [5], capacity shortages or exhaustions due to unpredictable surge in user traffic [6], and correlated fault in the underlying infrastructure [7, 8] are factors that have been reported to induce undesirable performance behaviour and service outages.

Performance variability and degradation impact the reliability of infrastructure and services and may hinder the ability to meet both service-level and operational objectives. Improving service performance and reliability is thus important since the volume of service traffic and the revenue relates to the quality of service (QoS) experienced by end-users [8]. Here are some concrete examples. The Aberdeen Group [9] found that a one-second increase in page response time can decrease page views, end-user satisfaction, and revenue by 11%, 16%, and 7%, respectively. On June 29th 2010, Amazon.com suffered a 3-hour intermittent performance problems such as high page latency and

incomplete product search results, which led to a loss of $1.75m in revenue per hour [10]. On the 16th of October 2015, several Apple services including iTunes and App Store suffered outages spanning many hours. Other popular global services such as Twitter, Netflix, Instagram, Google services (Drive, Docs, and Sheets), and PayPal also experienced hours of unplanned and disruptive outages in the same month [11]. Even national services such as the Swedish Tax Agency (Skatteverket) has on different occasions experienced outages and poor performance during the yearly income tax declarations [12, 13]. Besides revenue loss and damaged reputation, enterprises spend many work-hours diagnosing and restoring services[14].

Due to financial and operational implications of these incidents, performance diagnosis has thus become a major challenge in the day-to-day operation of cloud services and infrastructure. Operators must discover anomalous performance behaviour quickly, identify symptoms and root-causes, and deploy corrective strategies to remediate problems. The need to improve user experience, achieve stable performance, and reduce diagnosis time has given rise to many systems and methods. The contribution of this thesis is as follows. In Paper I we present an extensive survey of existing systems and methods. The aims are a) to classify existing methods based on their objectives and methodologies, and b) to assess research trends and open challenges. Paper III elaborates on a system for identifying capacity bottlenecks in a logical graph of datacenter components. It is presented as a use-case within Intel's Apex Lake framework. The Apex Lake is framework is being developed at the Cloud Service Labs, Intel Ireland and is introduced in Paper II. Finally, Paper IV proposes two adaptive mechanisms for detecting point anomalies in real-time performance behaviour of a web application in a cloud environment.

The remainder of this thesis introduction is organized as follows. Chapter 2 introduces the general concepts of cloud computing and challenges for achieving good performance. Chapter 3 discusses concepts in performance management and diagnosis in cloud datacenters. Chapter 4 summarizes the contributions of this thesis and discusses opportunities for future work. Papers produced in the course of this work are appended.

Chapter 2

Cloud Computing

Cloud computing is a paradigm shift in the way computation is delivered and consumed. It is based on the vision that information processing can be done more efficiently on large farms of computing and storage systems accessible via the Internet [15]. It provides as set of technologies that allow computational resources (e.g. compute and storage) to be provisioned *as a service* over the Internet on a pay-per-use and on-demand manner [16]. The growing adoption rate of the cloud is due to two appeal factors a) it offers seamless scalability and elasticity to users without the huge initial capital expenditure b) resources that can be metered so users are billed for only what they used [3]. These have led to a radical change in the way IT departments and application providers deploy and manage IT services [17].

2.1 Clouds Deployment Models

Cloud infrastructures can be classified based on ownership and how they are managed.

- *Private clouds* are infrastructures managed and utilized by individual organizations. Many organizations adopt private clouds to gain the benefits of cloud computing (such as flexibility and scalability) while having full control on the infrastructure and its security [15].
- *Public clouds* are virtualized infrastructures managed by an organization but leased on a pay-per-use basis to third-parties e.g. Google Cloud Platform, Amazon Web Services, Microsoft Azure, and RackSpace.
- *Community clouds* are shared by group of organizations or individuals with a common interest (e.g. mission, security, regulatory and compliance considerations). The North Carolina Education Cloud (NCEdCloud) is a community cloud [18].

- *Hybrid clouds* allow large enterprises who typically host their core IT infrastructure (compute and storage) on private or community clouds to team up with at least a public cloud for the purpose of extending the capacity of their private clouds arbitrarily by leasing public cloud resources to meet sudden surge in user demand through a mechanism known as *cloud bursting* [3].

- *Federated clouds* are an emerging type of clouds composed of only public clouds, only private clouds or both to provide seemingly infinite service computing utility [19, 20] to end-users. A typical instance of federated clouds is one established among multiple datacenters owned by a single cloud provider. Federation enables high compatibility and interoperability between disparate cloud services through open APIs that allows cloud users to deploy services based on offerings from different vendors or move data easily across platforms.

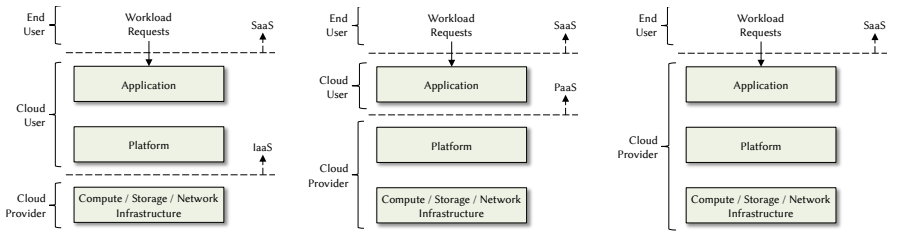
2.2 Cloud Delivery Models

Public clouds offer three major service delivery models based on the level of abstraction at which the service is offered [3, 21, 15].

- *Software as a Service* (SaaS)– delivery of finished applications, along with required software, operating system, network and hardware, to end-users and accessible through various client devices over the Internet. Though it is has minimal customization, the users generally have limited control on the application, platform and underlying infrastructure. SaaS applications include Customer Relationship Management (e.g. Salesforce, and ServiceNow), accounting/ERP systems (e.g. NetSuite) and Web 2.0 applications (e.g. LinkedIn, and WordPress).

- *Platform as a Service* (PaaS)—provides the capability for cloud users to build, deploy and manage applications using application development tools, APIs, operating systems, and hardware supported by the provider. The user only has control on the application, its architecture and hosting environment configuration but not on the underlying operating system, network, storage, or servers. Google App Engine, Windows Azure, and Amazon's Elastic Beanstalk are some of the most popular PaaS offerings today.

- *Infrastructure as a Service* (IaaS)—provisioning of fundamental computing resources such as servers, network, and storage; users are responsible for the installation of the operating system and can use application development tools of choice to develop arbitrary applications. Examples of IaaS providers are Amazon EC2, Google Cloud Platform, Microsoft Azure Infrastructure and Rackspace.



(a) IaaS (b) PaaS (c) SaaS

Figure 1: Roles of stakeholders under the three service delivery models [3].

2.3 Cloud Stakeholders, Roles and SLA

Stakeholders in cloud computing environments are generally categorized into one of three roles depending on the service delivery models [3]. According to Fig. 1, the responsibility of the *Cloud Provider* or Infrastructure Provider (IP) varies from provisioning only infrastructures in IaaS, provisioning both platforms and infrastructures in PaaS, to provisioning entire cloud stack (infrastructure, platforms, and application) in SaaS. A *Cloud User* or *Service Provider* (SP) is prominent under the IaaS and PaaS model. The service provider uses infrastructure and/or platform provided by the IP to host application services that are consumed by the End User. An End User generates application workloads that are processed by a cloud providers resources in the case of SaaS, or by a service provider in the case of PaaS and IaaS.

Interactions between the roles, especially, between the cloud and service providers may be governed by a formal contract document called the Service Level Agreement (SLA). An SLA describes expected quality of service (QoS) and legal guarantees [22]. A typical SLA contains, among others, important items such as the *service guarantees* and *service credits*. The service guarantees specifies functional metrics (e.g. availability, response time, and security) which a cloud provider strives to meet over a service guarantee time period. The service credit is an amount credited to the customer or applied towards a future payment when a cloud provider could not meet service guarantees [23]. For example, Amazon EC2, currently offer between 99% to 99.95% service availability guarantees with up to 10% of original service rate as service credits [24].

2.4 Cloud Datacenters

Cloud computing is powered by datacenters housing several servers, communications and storage systems co-located because of their common environmental, physical security, and maintenance requirements [8]. Consolidation is a method

used in cloud datacenters to ensure efficient utilization of these resources. By sharing server resources among multiple applications cloud providers prevent under-utilization and server sprawl in datacenters [10, 8].

Virtualization technology makes consolidation and sharing better by providing performance isolation among co-located applications. It is also cost-effective since consolidation reduces capital and operational expenses as well as energy consumption [25]. The two mainstream platforms for realizing virtualization today are *hypervisor-based* virtualization and *container-based* virtualization [26]. Hypervisor-based virtualization emulates a machine hardware and allows instances of the emulation (i.e., *virtual machines* (VMs)) to run on another physical machine managed by a specialized operating system (OS)–called the *hypervisor*. This approach is OS agnostic since the *guest* OS (i.e., OS installed on the VM) may differ from the *host* OS (i.e., OS running on the physical host of the VM). Xen [27], KVM [28], and Hyper-V [29] are examples of popular hypervisors [15, 30]. Rather than emulating an hardware platform, container-based virtualization provides virtualization at the OS-level in order to reduce performance and speed overhead of hypersor-based virtualization [31]. OS-level virtualization allows multiple isolated Linux environments (i.e., *containers*) to share the base kernel of the host OS [26, 32]. Docker [33, 34], LXC [35], and OpenVZ [36] are examples of container-based virtualization platforms.

Though containers are now enjoying growing adoption, VMs remain the common unit of cloud deployments due to the maturity of the technology. VMs are manageable software-defined units and facilitate the elasticity of the cloud. They can be easily moved (*migration*) from one server to another in order to balance the load across the datacenter or to consolidate workloads on fewer servers. They can also be replicated across servers (*horizontal scaling*) and their resource capacity (e.g. CPU cores) can be increased or decreased to address overload or underload situations (*vertical scaling*).

A major challenge in IaaS datacenters is resource allocation, which is concerned with how to optimally share or allocate computing, storage and network resources among a set of VMs in a manner that attempts to jointly meet service objectives of both the service providers and cloud provider [3, 37]. This has been identified as a difficult task [38, 39] due to the scale of datacenters, the plurality of applications and their complex architectures.

2.4.1 Software-defined Infrastructures

The growing adoption of cloud computing for running mission-critical and performance-sensitive applications such as analytics [40], machine-to-machine communications [41], mobile services [42] and the IoT [1] [43] is driving a higher demand for performance, agility, cost and energy efficiency. The need for increased automation and programmability of the infrastructure to meet these

[1] IoT is an acronym for Internet of Things, a term used to refer to the ever-increasing network of physical objects through the Internet, Radio Frequency IDentification (RFID) and other sensor network technologies.

requirements has led to recent trend towards Software-defined Infrastructures (SDI).

Many studies have applied *autonomic computing* [44] techniques in various aspects of managing existing clouds such as resource allocation and scheduling [45, 46, 47] as well as energy management [48, 49, 50] to facilitate automation. SDIs are envisioned to transform the compute, storage and network infrastructures to software-defined and dynamically programmable entities [51]. According to Kandiraju et al [52] an SDI is an infrastructure that is continuously transforming itself by appropriately exploiting heterogeneous capabilities, using insights gained from built-in deep monitoring, to continuously honor consumer SLAs amidst provider's constraints (e.g. cost and energy constraints). While only a few SDI products or platforms are currently available–e.g. VMware's EVO:RAIL [53] and Google's Andromeda [54]–many aspects of the SDI technology are still evolving. However, the core architectural components of SDIs are described below;

- *Software-defined Compute* (SDC): increases the programmability of existing virtualized compute resources (e.g. VMs, containers, virtual CPUs, memory) and dynamically leverages specialized processing units such as graphical processing units (GPUs), field-programmable gate arrays (FPGAs), and other accelerators [52]. SDC decouples provisioning of heterogeneous compute resources from the underlying hardware or operating system so that provisioning is based on specified or discovered workload requirements [55].

- *Software-defined Network* (SDN): separates control and management functions of the network infrastructures away from the hardware to the server for improved programmability, performance isolation, efficiency and security [51]. Through virtualization, SDN allows network resources to be transformed to virtual devices (e.g. switches, links and end-points) that connects different virtual compute and storage instances [52].

- *Software-defined Storage* (SDS): is responsible for managing huge volume of data by isolating the control and management functions from the data storage system and dynamically leverages specialized storage when available [51]. The advantage of such isolation is that it reduces management complexity and also the cost of infrastructure [56]. The basic unit of a SDS is the virtual storage—an abstraction of a combination of different storage capabilities (e.g. access bandwidth (IOPS), solid-state disk, RAID, block storages, distributed file systems) to guarantee SLAs [52].

The SDI components are managed by a SDI *controller* that provides the control intelligence required to meet workload requirements and SLA. The SDI controller uses the classical MAPE loop [57] to continuously keep track of the current state of SDI entities and acts on virtual and physical infrastructure to ensure SLA compliance [52].

Chapter 3

Cloud Performance Management

The performance of a system can be defined in terms of the system's ability to perform a given task correctly. If it does, then its performance is measured by the time taken to perform the task, the rate at which the task is performed, and the resources consumed while performing the task [58]. Hence, performance is composed of two major components [58, 59] namely;

- *Time behaviour*: The degree to which the response time, execution time and throughput rates of a system meet requirements when performing its functions.
- *Resource utilization*: The degree to which the amounts and types of resources used by a system meet requirements when performing its functions.

In general, the efficiency of a system can thus be expressed in terms of performance. That is its time behaviour relative to its resource utilization under stated conditions. The most commonly used performance metrics are [58];

- *Response time*–is time interval between the end of a request submission and the end of the corresponding response from the system for interactive users in a timeshared system (also known as *latency*). For batch jobs, it is the time between the submission of a batch job and the completion of its output (also called *turnaround time*). The latency of a web request may be expressed in milliseconds or even in minutes.
- *Throughput*–is the rate (requests per unit of time) at which application requests can be serviced by a system. For a batch job, throughput may be expressed in terms of jobs/tasks completed per unit time. For interactive applications, throughput can be described in terms of the number of requests or transactions completed over a given period of time.

- *Utilization*–is the fraction of time a resource is busy servicing requests. A simple model of CPU utilization is to compute the ratio of busy time to total elapsed time over a given period (expressed in percentage).

Performance is related to the majority of obstacles for adoption and growth of cloud computing such as availability, capacity bottlenecks, scalability, reliability and security [60]. Hence, it is important for user satisfaction and guaranteed service assurance [61, 8]. Cloud computing relies on effective performance management to meet service-level objectives (SLO) and SLA. Performance management in a cloud environment involves the monitoring and measurement of relevant performance metrics of hosted application services and infrastructure in order to assess their performance. It also encompasses the analysis and visualization of the data, as well as detection and diagnosis of performance problems.

Performance management can be viewed from two perspectives depending on the of role a cloud stakeholder [62]. Figure 2 presents these perspectives in the context of an IaaS cloud.

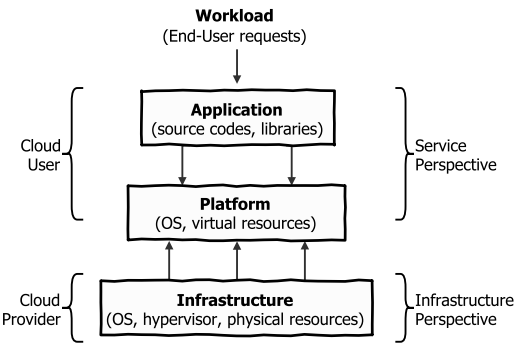


Figure 2: Performance perspectives in IaaS clouds.

The *service perspective* is a top-down approach to performance management and is in the purview of the service provider. It is concerned with performance and workload modeling and analysis for capacity planning (resource estimation), online resource control, anomaly detection, SLA monitoring, as well as end-user experience and behaviour learning. According to Gregg et al [62], the target metrics in this perspective includes requests rates–workload applied to the service, latency–responsiveness of the service, and completion–error rates. While the service provider generally has little to no knowledge about the external runtime environment, she has complete control on her service architecture, operating systems, source codes; and thus can perform drilled-down analysis.

The *infrastructure perspective* provides a bottom-up view of the entire infrastructure from physical infrastructure (compute, network, storage) through the virtual infrastructure (VMs, containers, etc) to the hosted services. Though the cloud provider has limited access to the behaviour and performance of hosted

applications and platforms, she is responsible for monitoring and analyzing the operational metrics of physical infrastructure components. The cloud provider must ensure that statistically multiplexing limited hardware resources among disparate applications black-boxes (VMs) do not affect the SLA and performance of those applications. Resource analysis is a major task in this perspective. It involves analysis of performance metrics of resources such as CPUs, memory, disks, network interfaces, buses and interconnects [62]. Results of the analysis are applicable in two associated activities a) performance diagnosis–identifying resource bottlenecks, faults or other root-causes to explain reported service-level anomalies or SLA violations, and b) capacity planning–this includes resource demand profiling, resource estimation, and resource scaling. Resource throughput, utilization, and saturation are commonly used metrics in this case.

The perspectives also affect how datacenter components are monitored. The service perspectives allows activities such as request/transaction tracing or tagging, source-code instrumentation, profiling of virtual resources by the service provider; while the cloud provider can only perform infrastructure monitoring such as physical resource profiling and kernel instrumentation in the host operating system. The monitoring aspect of performance management in cloud computing is treated extensively in [63, 64]. The focus of this thesis is on analyzing performance metrics data for diagnosing performance problems in cloud infrastructure.

3.1 Cloud Performance Diagnosis

The occurrence of performance issues is a norm, rather an exception in large scale systems [65]. Performance anomalies and unexpected variability are common even in large-scale infrastructures such as in Google clusters [5] and Amazon Web Services [4]. Apart from damaging end-user experience and service reputation, performances problems sometimes lead to unplanned outages. The far-reaching consequences of performance problems drives the need for techniques and systems to automatically detect problems and guide operators to potential root-causes and solutions.

In general, the goal of performance diagnosis in a datacenter is (a) to detect both expected and unexpected performance events, (b) analyze operational metrics to gather evidences for localizing root-cause(s) responsible for observed behaviour, and (c) to either recommend strategies that may resolve the problem or automatically actuate the remediation.

From the service performance perspective, this thesis deals only with post-deployment diagnosis (i.e. during operation). Pre-deployment diagnosis of performance anti-patterns when testing for software quality during development phase [66, 67, 68] is beyond the scope of this work.

3.1.1 Challenges

The following are cloud-specific factors that we consider as obstacles to effective performance diagnosis in datacenters.

1. *Scale*: A single datacenter may host up to tens of thousands servers. Multiple datacenters owned by a large vendor such as Microsoft or Google may contain up to one million servers in total from multiple datacenters [69]. Through consolidation, each server can host tens or even hundreds of VMs and applications.

2. *Complexity*: The topology of the cloud infrastructure is complex; they span single facility to geographically dispersed facilities. Cloud datacenters support a heterogeneous mix of application services ranging from latency-sensitive services (e.g. multi-tier web applications) to batch-oriented services (e.g. scientific simulations). Also, applications often have varying and sometimes conflicting performance objectives. Diagnosing problems in this environment is further aggravated by limited observability due to separation of control between cloud and service providers.

3. *Dynamism*: Cloud applications contain multiple inter-dependent components sharing equally inter-dependent physical resources. The ensuing complex dependency is bound to induce dynamic behaviours and cascading bottleneck or faults. While co-location of multiple applications on the same server improves datacenter resource and energy utilization, it has been reported to cause severe variability in performance of latency-sensitive services [5] due to lack of performance isolation. Due to dynamic resource management, the execution context of services is also constantly changing as load continuously flow in and out of the datacenter.

4. *Autonomics*: Datacenters have become highly autonomous since manual management is not practical from a performance perspective [70]. Such advancement require performance diagnosis systems that can learn and operate in on-line scenario. Offline analysis and detection, as well as ticket-based manual resolution may not fit this environment.

5. *Monitoring and the data deluge*: The performance of services and infrastructure is currently monitored using diverse tools [62]. The resulting data end up in varying format and semantics with noises that sometimes mask anomalies. Also, datacenters produce a staggering amount of streaming operational data [70] that can easily overwhelm operators and alert systems.

6. *Complex problem structure*: The notion of performance is generally subjective [62]. What constitute normal or anomalous behaviour varies from service to service and from one infrastructure provider to another. Also, the occurrence of anomalies is dynamic. Some anomalies are highly contextual because they occur under specific workload and usage situations.

There could be a wide variety of potential root-cause even for a minute drop in latency. Each problem has a characteristic behaviour visible only by certain performance metrics. Some problem symptoms are recurrent in nature; so eliminating a manifestation may not necessarily eliminate the actual root-cause nor solve the problem [68].

3.1.2 Categories of performance problems

Wert et al [68] proposed that performance problems can be categorized into three layers namely; *symptoms*–externally visible indicators of a performance problem, *manifestation*–internal performance indicators or evidences, and *root-causes*–physical factors whose removal thereof eliminates the manifestations and symptoms of a performance incident. We adopt their classification to categorize common performance problems in Table 3.1.

3.1.3 Activities in performance diagnosis

Performance diagnosis activities can be broadly divided into two subtasks:

- *Identification*: The goal of performance problem identification is two-fold. First task is to, in a timely manner, discover performance symptoms (e.g. degraded response time) and/or deviations from expected behaviour based on thresholds, baselines, models, or SLAs. This is known as Performance Anomaly Detection (PAD). SEAD [71], EbAT [72], PAD [73], and UBL [74] perform this task. Once a symptom is detected, next task is to figure out if it is similar to a previously seen problem so that a known root-cause and solution can be recommended. This speeds up the diagnosis process and avoids escalation [65]. If it is a new symptom, the system must find evidences or manifestations that partly explains the anomaly and help narrow search path for potential root-causes. Fingerprinting is a common technique for summarizing and persisting previously seen problems [65]. Example systems includes Spectroscope [75], Prepare [76], vPerfGuard [77], PerfCompass [78], and Orion [79].
- *Root-cause analysis*: This involves analysis of identified symptoms (anomalies) and manifestations to determine the potential root-causes and suggesting actions to correct them. Accurate localization of the problem is important to mitigate performance problems, restore the system to good state, and for future recall. Examples of such systems include X-ray [80], PerfScope [81], PeerWatch [82].

In addition to problem diagnosis, performance management also include the *remediation* of performance problems. This involves taking actions to restore the system to normal state. It relies on information and recommendations from the two phases of the diagnosis process. Systems such as CloudPD [83] and CPI2 [5] propose strategies for correcting performance problems due to capacity related causes.

Table 3.1: Classification of common systems performance problems.

Problem	**Category**	**Perspective**	**Instances**
Increasing response time	Symptoms	Service	Non-responsive application, Unexpected slow down in file transfer, Degraded video quality or resolution.
Throughput degradation under load	Symptoms	Service	Dropped requests or packets, Incomplete search results, Degraded video resolution.
Service unavailability	Symptoms	Service	Service hanging or crashing under heavy load, Unreachable service.
SLA violation	Symptoms	Service	Availability $< 99\%$, Latency $> 3ms$.
Sudden latency spikes	Symptoms	Service	Latency $>$ ($mean$ latency $+ 6sd$)
Resource under-utilization under load	Manifestation	Service/Infrastructure	Low IO utilization under heavy IO workload.
Resource over-utilization	Manifestation	Service/Infrastructure	High CPU utilization (near 100%)
High resource saturation	Manifestation	Service/Infrastructure	100% CPU utilization High average resource queue length, Excessive dropping of network packets.
Resource contention or interference	Manifestation	Service/Infrastructure	High CPU steal time, High cycle-per-instruction (CPI).
Resource dependency	Manifestation	Service/Infrastructure	High CPU waiting time due to IO
Network congestion	Root-cause	Service/Infrastructure	Packet loss, Queueing delays, Blocking of new connections.
Unexpected workload spikes	Root-cause	Service/Infrastructure	Sudden spike in update requests, Flash-crowd behaviour.
Periodic load behaviour	Root-cause	Service/Infrastructure	Expected workload fluctuation, Diurnal workload pattern.
Application-level issues	Root-cause	Service/Infrastructure	Component or dependency failure, Bugs in application codes, Exhausted thread pools, Failed application upgrade, Corrupted data error
Capacity bottlenecks	Root-cause	Service/Infrastructure	Resource hogging and contention, Memory leaks, Unanticipated interference.
Platform & OS issues	Root-cause	Service/Infrastructure	Transient events e.g. Memory hardware errors, JVM garbage collector issues.
Machine-level faults	Root-cause	Infrastructure	Faulty server components, DRAM soft-errors or disk errors, OS kernels bugs, abnormal reboots, Unplanned power outage or overheating.
Routine system maintenance	Root-cause	Service/Infrastructure	Routine backups, Forklift replacement, OS upgrade.
Operator errors	Root-cause	Service/Infrastructure	Misconfiguration error, Inadvertent database locks, Erroneous overwrite or data deletion, Wrong price setting.
Security violations	Root-cause	Service/Infrastructure	Denial of service attacks, Worms and virus infection, or Theft.

Chapter 4

Summary of Contributions

This thesis contributes to the understanding of performance problem diagnosis in systems especially in cloud environments and also propose new techniques to improve the process. Firstly, we carried out an extensive survey of the research area, covering the classification and review of different approaches. We found that application-level performance diagnosis still rely on simplistic threshold-based techniques. Considering the impact of performance problems on SLA/SLO compliance, it is surprising why this is the case. Hence, we propose techniques for automatically detecting anomalies (symptoms) in real-time service performance behaviour in order to improve end-user experience as well as prevent SLA violations or service outages. Furthermore, we devised a knowledge-driven technique for identifying performance problem manifestations in datacenter resources within the context of Apex Lake[1]. The approach is based on combining certain performance metrics with the logical dependencies across resources. The significance of the work in this thesis is to drive performance diagnosis to be more automated improve the quality of service, user experience as well as the speed of diagnosis. The thesis contributions also include scientific publications and software prototypes.

Research Methodology. The methodology used in this thesis is experimental. We begin by definition of problems, literature review, design and implementation of algorithms, design of experiments and analysis of results. There are generally two evaluation approaches in this area. The first is to evaluate proposed algorithms using public datasets. The other is through active perturbation [84] whereby algorithms are evaluated empirically on a real testbed with realistic load and fault injections to bring it close to the real environment being emulated. Though the former suffers from the lack of public application/systems performance data for broad use-cases, it is better suited for evaluating algorithms on independent data especially when the data is labeled. On the other hand, it is challenging to recreate desired behaviour in the later.

[1]Apex Lake is a framework developed at Intel for enhancing smarter orchestration in clouds especially in SDIs.

The later approach is taken to evaluate techniques proposed in Paper III and IV. We collect data from actual applications running in a virtualized testbed with emulated capacity bottlenecks via active perturbation.

4.1 Paper I

The ubiquity of performance problems has led to many interesting diagnosis techniques. However, there exist no classification of the objectives and characteristics of proposed approaches. In order to assess progress, identify trends and open challenges we conducted a survey of over forty research articles spanning over ten years (2002–2014) covering systems from a wide range of application domains (e.g. dedicated environments, distributed and grid systems, as well as multi-tier web services).

The survey, published by the ACM Computing Survey, gives a background of aspects of the problem, and categorizes existing solutions using multiple factors such as goals, domain of application, and methodology. Research trend shows that research volume and complexity of detection methodologies seem to follow the increasing complexity of systems. The task of detecting performance anomalies (i.e. symptoms of problems) dominates research contributions in this area; accounting for 53% of reviewed works. It is followed by problem identification (identifying bottlenecks or root-causes to explain anomalies) which accounts for 29% of reviewed papers. Hybrid systems combining the two tasks account for 18% of reviewed papers. Existing techniques exploit many strategies such as static thresholding, fine-grain workload models, fingerprints, flow paths or dependencies, expert knowledge (e.g. rules). Though statistical models and tests are popularly used, availability of cheap computing power has led to increasing adoption of machine learning techniques. While end-to-end, autonomous, and distributed diagnosis remain open challenges, the lack of open-source performance data and problem taxonomy hinder the pace of performance diagnosis research.

4.2 Paper II & III

Landscape colouring was introduced as a use-case within Intel's `Apex Lake` framework introduced in Paper II. While Apex Lake is being developed by Cloud Services Lab (CSL), Intel Labs Europe, Ireland, since 2014, Umeå University worked together with CSL to collaboratively develop and write Paper II as well as proposed the concept of landscape colouring as a use case for Apex Lake. Paper III refines and extends the original idea of landscape colouring as well as demonstrates it with preliminary experiments.

This work is motivated by the concerns of low datacenter utilization [8] and the need for tighter orchestration [51] in modern datacenters, such as software-defined environments, where the logical topology and composition of the datacenter is dynamic. We propose landscape colouring, a technique that

combines dynamic topology information with operational metrics of individual datacenter components (applications, servers, VMs, etc.) in order to reason about anomalies (or manifestations) such as resource over-utilization and saturation. We employed components of Apex Lake as follows. The landscaper is used to automatically and continuously organize the datacenter into a three-layer graph structure of service, virtual and physical entities. `Cimmaron` monitors operational metric, such as node (utilization) and the amount of tasks on its queue (saturation), that are used to characterize node behaviour into discrete operational states using a fuzzy inference system. Nodes are flagged anomalous if they persist in a specified set of undesirable states. When many nodes are flagged simultaneously, we rank anomalous nodes according to how important they are in the graph to enhance the resolution process. The rank score is a function of the influence of a node's neighbours, the layer it belongs, and the number of anomalous nodes depending on it.

We performed two experiments to demonstrate our approach in a virtualized environment with emulated CPU and IO saturation bottlenecks. Results show that the technique is able to identify and rank nodes with anomalies (e.g CPU contentions due to inter-node dependency). We plan to perform more experiments with multiple applications on a larger testbed (where the topology is truly dynamic) to investigate the scalability of technique and to correlate service-level symptoms with capacity bottlenecks in the infrastructure. We will also be exploring how the technique can be used to trigger corrective actions automatically or to drive other datacenter optimizations within the Apex Lake framework.

4.3 Paper IV

Application performance degradation (e.g. high latency) is common on the Internet today; and have been reported to be mostly due to capacity-related issues rather than application bugs or machine-level faults [65, 8]. However, it has received less research attention and existing studies are largely based on simplistic threshold-based techniques.

Due to the correlation between application performance and end-user satisfaction, we proposed and evaluated two schemes for detecting point anomalies in real-time service latency measurements. The first exploited the fact that applications sometimes have known baselines. We devised a behaviour-based anomaly detection technique using adaptive Kernel Density Estimation (KDE) and its KD-tree variants to detect deviations from given baseline. The second scheme is for the case where a known baseline behaviour does not exist, we proposed a prediction-based method to discover point anomalies. Significant deviations in prevailing service behaviour are detected by predicting service latency using Holt-Winter's forecasting and applying adaptive EWMA control chart limits on the prediction errors.

We evaluated the schemes using a web application benchmark on a virtualized

testbed with realistic load and bottleneck injections. We observed that the tree variant of the adaptive KDE algorithm offers lower error rates than the standard KDE. Under appropriate parameter settings, the prediction-based scheme is able to follow the trends in service behaviour and detect out-of-control anomalies. Results suggest that many service performance anomalies could be explained by system-level capacity bottlenecks but not all low-level bottlenecks lead to service-level anomalies. In the future, we plan to further evaluate accuracy of the techniques using labeled datasets from public system traces.

4.4 Future Work

This work can be extended on multiple fronts. In general, we aim to improve the evaluation of the proposed techniques. Though it is hard to perform substantial statistical evaluations of the anomaly detection using an experimental approach, we hope to accomplish this when we address root-cause analysis and problem remediation. We plan to evaluate techniques introduced in Paper IV using labeled datasets from real systems logs so that we can quantify their accuracy and false alarm rates. The experiments in Paper III are preliminary, we plan to assess the scalability of proposed method on a larger testbed with multiple applications; and to quantify accuracy with suitable statistical analysis.

Furthermore, the prediction-based anomaly detection technique will be extended with low-level resource metrics. By modeling relationships between application-level metrics and resource metrics, we may be able to pinpoint bottleneck resources to explain an observed anomaly. The static thresholds of the fuzzy membership functions in Paper III could be learned on-line to reflect prevailing workloads in the system. Also, in addition to ranking anomalous node, it could adapt the method to differentiate victim nodes from antagonist nodes when there is a resource contention.

Most performance problems are recurrent in nature [68] with well known causes. We shall attempt to structure domain knowledge about capacity-related anomalies and bottleneck in cloud infrastructure. By linking symptoms to manifestations and to root-causes, we posit this will contribute towards the realization of automated problem remediation since it will be easy to associate root-causes with potential remediation strategies. For example, upon detecting a latency spike (symptom), the fuzzy inference system may find CPU utilization to be consistently high (manifestation). Further analysis may expose unexpected workload spike (root-cause) as the actual culprit. A possible solution would be to scale CPU resource to meet the workload demand.

The overall goal of our research is to close the performance problem management loop; from problem identification through root-cause analysis to problem remediation. Specifically, we plan to investigate existing strategies and develop techniques for automatically rectifying observed problems. Such a technique could, for instance, adjust number of replicas or CPU cores of a VM to correct sustained latency growth caused by a workload spike or CPU bottleneck. On the

other hand, it could migrate a noisy VM to an idle host or enforce policy-based resource throttling, to correct throughput degradation in a batch job.

Bibliography

[1] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud Computing and Grid Computing 360-degree Compared," in *Grid Computing Environments Workshop, 2008. GCE'08*, pp. 1–10, IEEE, 2008.

[2] Q. Guan, C.-C. Chiu, and S. Fu, "CDA: A Cloud Dependability Analysis Framework for Characterizing System Dependability in Cloud Computing Infrastructures," in *18th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 11–20, IEEE, 2012.

[3] B. Jennings and R. Stadler, "Resource Management in Clouds: Survey and Research Challenges," *Journal of Network and Systems Management*, vol. 23, no. 3, pp. 567–619, 2015.

[4] A. Iosup, N. Yigitbasi, and D. Epema, "On the Performance Variability of Production Cloud Services," in *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 104–113, IEEE, 2011.

[5] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI 2: CPU Performance Isolation for Shared Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 379–391, ACM, 2013.

[6] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building more Robust Cloud Applications," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 700–711, ACM, 2014.

[7] C. Klein, A. V. Papadopoulos, M. Dellkrantz, J. Durango, M. Maggio, K.-E. Arzen, F. Hernández-Rodriguez, and E. Elmroth, "Improving Cloud Service Resilience using Brownout-aware Load-balancing," in *33rd International Symposium on Reliable Distributed Systems (SRDS)*, pp. 31–40, IEEE, 2014.

[8] L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[9] Aberdeen Group, "The Performance of Web Applications: Customers are Won or Lost in One Second (Research Report)." http://www.aberdeen.com/research/5136/ra-performance-web-application/content.aspx, May, 2015. Accessed: 2016-03-31.

[10] C. Wang, S. P. Kavulya, J. Tan, L. Hu, M. Kutare, M. Kasick, K. Schwan, P. Narasimhan, and R. Gandhi, "Performance Troubleshooting in Data centers: An Annotated Bibliography?," *ACM SIGOPS Operating Systems Review*, vol. 47, no. 3, pp. 50–62, 2013.

[11] CloudEndure, "The Top 9 Outages that made Headlines in Q4 2015 : CloudEndure." https://www.cloudendure.com/blog/top-9-outages-made-headlines-q4-2015/, 2015. Accessed: 2016-03-31.

[12] Göteborgs-Posten, "E-deklarationer överbelastade Skatteverket - Sverige - Göteborgs-Posten." http://www.gp.se/nyheter/sverige/1.2703888-e-deklarationer-overbelastade-skatteverket, 2015. Accessed: 2016-04-03.

[13] Aftonbladet, "Nyfikna skattebetalare sänkte sajten." http://www.aftonbladet.se/nyheter/article22488308.ab, 2016. Accessed: 2016-04-03.

[14] Evolven Software, "Downtime, Outages and Failures–Understanding Their True Costs." http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html, September, 2011. Accessed: 2016-04-18.

[15] D. C. Marinescu, "Cloud Computing: Theory and Practice." http://www.scopus.com/inward/record.url?eid=2-s2.0-84902054996&partnerID=tZOtx3y1, 2013.

[16] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud Computing: State-of-the-art and Research Challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[17] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[18] EDUCAUSE, "Spotlight on Cloud Computing: Community Clouds." https://net.educause.edu/ir/library/pdf/LIVE1017b.pdf, 2010. Accessed: 2016-04-01.

[19] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, *et al.*, "The Reservoir Model and Architecture for Open Federated Cloud Computing," *IBM Journal of Research and Development*, vol. 53, no. 4, pp. 4–1, 2009.

[20] B. Rochwerger, A. Galis, E. Levy, J. A. Caceres, D. Breitgand, Y. Wolfsthal, I. M. Llorente, M. Wusthoff, R. S. Montero, and E. Elmroth, "Reservoir: Management Technologies and Requirements for Next Generation Service Oriented Infrastructures," in *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, pp. 307–310, IEEE Press, 2009.

[21] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *National Institute of Standards and Technology*, vol. 53, no. 6, p. 50, 2009.

[22] M. Alhamad, T. Dillon, and E. Chang, "Conceptual SLA Framework for Cloud Computing," in *4th IEEE International Conference on Digital Ecosystems and Technologies (DEST)*, pp. 606–610, IEEE, 2010.

[23] S. A. Baset, "Cloud SLAs: Present and Future," *ACM SIGOPS Operating Systems Review*, vol. 46, no. 2, pp. 57–66, 2012.

[24] Amazon, Inc., "Amazon EC2 Service Level Agreement." http://aws.amazon.com/ec2/sla/. Accessed: 2016-04-04.

[25] G. Jung, M. A. Hiltunen, K. R. Joshi, R. D. Schlichting, and C. Pu, "Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures," in *30th International Conference on Distributed Computing Systems (ICDCS)*, pp. 62–73, IEEE, 2010.

[26] W. Li and A. Kanso, "Comparing Containers versus Virtual Machines for Achieving High Availability," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pp. 353–358, IEEE, 2015.

[27] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and The Art of Virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[28] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux Virtual Machine Monitor," in *Proceedings of the Linux symposium*, vol. 1, pp. 225–230, 2007.

[29] A. Velte and T. Velte, *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[30] N. Antonopoulos and L. Gillam, *Cloud computing: Principles, Systems and Applications*. Springer Science & Business Media, 2010.

[31] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 275–287, ACM, 2007.

[32] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pp. 171–172, IEEE, 2015.

[33] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, Mar. 2014.

[34] Docker, Inc., "Docker: The Linux Container Engine." https://www.docker.com/. Accessed: 2016-04-18.

[35] "LXC: LinuX Container." https://linuxcontainers.org/. Accessed: 2016-04-18.

[36] "OpenVZ Linux Containers." https://openvz.org/. Accessed: 2016-04-18.

[37] J. G. et al., "Survey on Cloud Computing Resource Allocation Models and Methods," *International Journal of Computer Science and Management Research*, vol. 1, December 2012.

[38] A. Gulati, G. Shanmuganathan, A. Holler, and I. Ahmad, "Cloud-scale Resource Management: Challenges and Techniques," in *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, pp. 3–3, USENIX Association, 2011.

[39] P. Xiong, "Dynamic Management of Resources and Workloads for RDBMS in Cloud: A Control-theoretic Approach," in *Proceedings of the on SIGMOD/PODS 2012 PhD Symposium*, pp. 63–68, ACM, 2012.

[40] D. Talia, "Toward cloud-based big-data analytics," *IEEE Computer Science*, pp. 98–101, 2013.

[41] I. Stojmenovic, "Fog computing: a cloud to the ground support for smart things and machine-to-machine networks," in *Telecommunication Networks and Applications Conference (ATNAC), 2014 Australasian*, pp. 117–122, IEEE, 2014.

[42] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 84–106, 2013.

[43] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[44] M. C. Huebscher and J. A. McCann, "A Survey of Autonomic Computing—degrees, Models, and Applications," *ACM Computing Surveys (CSUR)*, vol. 40, no. 3, p. 7, 2008.

[45] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service," in *Proceeding of the USENIX International Conference on Automated Computing (ICAC'13). San Jose, CA*, 2013.

[46] H. Nguyen Van, F. Dang Tran, and J.-M. Menaud, "Autonomic Virtual Resource Management for Service Hosting Platforms," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2009.

[47] A. Roytman, A. Kansal, S. Govindan, J. Liu, and S. Nath, "PACMan: Performance Aware Virtual Machine Consolidation," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, (San Jose, CA), pp. 83–94, USENIX, 2013.

[48] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, and J. L. Hellerstein, "Dynamic energy-aware capacity provisioning for cloud computing environments," in *Proceedings of the 9th international conference on Autonomic computing*, pp. 145–154, ACM, 2012.

[49] M. Guazzone, C. Anglano, and M. Canonico, "Energy-efficient resource management for cloud computing infrastructures," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pp. 424–431, IEEE, 2011.

[50] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 13–23, ACM, 2007.

[51] C. Li, B. Brech, S. Crowder, D. M. Dias, H. Franke, M. Hogstrom, D. Lindquist, G. Pacifici, S. Pappe, B. Rajaraman, *et al.*, "Software defined Environments: An Introduction," *IBM Journal of Research and Development*, vol. 58, no. 2/3, pp. 1–1, 2014.

[52] G. Kandiraju, H. Franke, M. Williams, M. Steinder, and S. Black, "Software defined Infrastructures," *IBM Journal of Research and Development*, vol. 58, no. 2/3, pp. 2–1, 2014.

[53] VMware, Inc., "EVO:RAIL Hyper-Converged Infrastructure Appliance." http://www.vmware.com/products/evorail/. Accessed: 2016-04-19.

[54] Google, Inc., "Enter the Andromeda zone - Google Cloud Platform's Latest Networking Stack." https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html. Accessed: 2016-04-19.

[55] Guru Rao, "Software Defined Compute Provides Workload-aware Infrastructure and Optimization through Automation and Open Technologies." https://www.ibm.com/developerworks/community/blogs/ibmsyssw/entry/software_defined_compute_provides_workload_aware_infrastructure_and_optimization_through_automation_and_open_technologies?lang=en, January, 2014. Accessed: 2016-04-05.

[56] Y. Jararweh, M. Al-Ayyoub, E. Benkhelifa, M. Vouk, A. Rindos, *et al.*, "Software defined Cloud: Survey, System and Evaluation," *Future Generation Computer Systems*, 2015.

[57] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[58] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, New York, 1991.

[59] L. Bautista, A. Abran, and A. April, "Design of a Performance Measurement Framework for Cloud Computing," 2012.

[60] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "Above the Clouds: A Berkeley View of Cloud Computing," *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, no. 13, p. 2009, 2009.

[61] Shailesh Paliwal, "Performance Challenges in Cloud Computing." https://www.cmg.org/wp-content/uploads/2014/03/1-Paliwal-Performance-Challenges-in-Cloud-Computing.pdf, 2014. Accessed: 2016-04-05.

[62] B. Gregg, *Systems Performance: Enterprise and the Cloud*. Pearson Education, 2014.

[63] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, "Cloud monitoring: A Survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.

[64] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn, "A survey of Cloud Monitoring Tools: Taxonomy, Capabilities and Objectives," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2918–2933, 2014.

[65] H. Malik and E. M. Shakshuki, "Classification of Post-deployment Performance Diagnostic Techniques for Large-scale Software Systems," *Procedia Computer Science*, vol. 37, pp. 244–251, 2014.

[66] C. Trubiani and A. Koziolek, "Detection and Solution of Software Performance Antipatterns in Palladio Architectural Models," in *ACM SIGSOFT Software Engineering Notes*, vol. 36, pp. 19–30, ACM, 2011.

[67] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "Mining Performance Regression Testing Repositories for Automated Performance Analysis," in *10th International Conference on Quality Software (QSIC)*, pp. 32–41, IEEE, 2010.

[68] A. Wert, "Performance Problem Diagnostics by Systematic Experimentation," in *Proceedings of the 18th international doctoral symposium on Components and architecture*, pp. 1–6, ACM, 2013.

[69] StorageServers, "Facts and Stats of Worlds Largest Data centers." https://storageservers.wordpress.com/2013/07/17/facts-and-stats-of-worlds-largest-data-centers/, July 2013. Accessed: 2016-04-07.

[70] J. Murphy, "Performance Engineering for Cloud Computing," in *Computer Performance Engineering*, pp. 1–9, Springer, 2011.

[71] H. S. Pannu, J. Liu, and S. Fu, "A Self-evolving Anomaly Detection Framework for Developing Highly Dependable Utility Clouds," in *Global Communications Conference (GLOBECOM), 2012 IEEE*, pp. 1605–1610, IEEE, 2012.

[72] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online Detection of Utility Cloud Anomalies using Metric Distributions," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pp. 96–103, IEEE, 2010.

[73] M. Peiris, J. H. Hill, J. Thelin, S. Bykov, G. Kliot, and C. Konig, "PAD: Performance Anomaly Detection in Multi-server Distributed Systems," in *7th International Conference on Cloud Computing (CLOUD)*, pp. 769–776, IEEE, 2014.

[74] D. J. Dean, H. Nguyen, and X. Gu, "UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems," in *Proceedings of the 9th international conference on Autonomic computing*, pp. 191–200, ACM, 2012.

[75] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing Performance Changes by Comparing Request Flows," in *NSDI*, 2011.

[76] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems," in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pp. 285–294, IEEE, 2012.

[77] P. Xiong, C. Pu, X. Zhu, and R. Griffith, "vPerfGuard: An Automated Model-driven Framework for Application Performance Diagnosis in Consolidated Cloud Environments," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pp. 271–282, ACM, 2013.

[78] 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*, pp. 16–16, USENIX Association, 2014.

[79] I. Laguna, S. Mitra, F. A. Arshad, N. Theera-Ampornpunt, Z. Zhu, S. Bagchi, S. P. Midkiff, M. Kistler, and A. Gheith, "Automatic Problem Localization via Multi-dimensional Metric Profiling," in *32nd International Symposium on Reliable Distributed Systems (SRDS)*, pp. 121–132, IEEE, 2013.

[80] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pp. 307–320, 2012.

[81] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang, "Perfscope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–13, ACM, 2014.

[82] H. Kang, H. Chen, and G. Jiang, "PeerWatch: A Fault Detection and Diagnosis Tool for Virtualized Consolidation Systems," in *Proceedings of the 7th international conference on Autonomic computing*, pp. 119–128, ACM, 2010.

[83] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, "CloudPD: Problem Determination and Diagnosis in Shared Dynamic Clouds," in *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, IEEE, 2013.

[84] R. Apte, L. Hu, K. Schwan, and A. Ghosh, "Look Who's Talking: Discovering Dependencies between Virtual Machines Using CPU Utilization," in *HotCloud*, 2010.

I

Paper I

Performance Anomaly Detection and Bottleneck Identification

Olumuyiwa Ibidunmoye, Francisco Hernandez-Rodriguez, Erik Elmroth

Dept. Computing Science, Umeå University
SE-901 87 Umeå, Sweden
{muyi, francisco, elmroth}@cs.umu.se
http://www.cs.umu.se/ds

Abstract: In order to meet stringent performance requirements, system administrators must effectively detect undesirable performance behaviours, identify potential root causes and take adequate corrective measures. The problem of uncovering and understanding performance anomalies and their causes (bottlenecks) in different system and application domains is well studied. In order to assess progress, research trends and identify open challenges, we have reviewed major contributions in the area and present our findings in this survey. Our approach provides an overview of anomaly detection and bottleneck identification research as it relates to the performance of computing systems. By identifying fundamental elements of the problem, we are able to categorize existing solutions based on multiple factors such as the detection goals, nature of applications and systems, system observability, and detection methods.

Performance Anomaly Detection and Bottleneck Identification

OLUMUYIWA IBIDUNMOYE, FRANCISCO HERNÁNDEZ-RODRIGUEZ,
and ERIK ELMROTH, Umeå University

In order to meet stringent performance requirements, system administrators must effectively detect undesirable performance behaviours, identify potential root causes, and take adequate corrective measures. The problem of uncovering and understanding performance anomalies and their causes (bottlenecks) in different system and application domains is well studied. In order to assess progress, research trends, and identify open challenges, we have reviewed major contributions in the area and present our findings in this survey. Our approach provides an overview of anomaly detection and bottleneck identification research as it relates to the performance of computing systems. By identifying fundamental elements of the problem, we are able to categorize existing solutions based on multiple factors such as the detection goals, nature of applications and systems, system observability, and detection methods.

Categories and Subject Descriptors: C.4 [**Computer-Communication Networks**]: Performance of Systems—*Reliability, availability, and serviceability*; D.4.8 [**Operating Systems**]: Performance—*Measurement, modeling and prediction*

General Terms: Performance, Reliability

Additional Key Words and Phrases: Systems performance, performance anomaly detection, bottleneck detection, performance problem identification

ACM Reference Format:
Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. 2015. Performance anomaly detection and bottleneck identification. ACM Comput. Surv. 48, 1, Article 4 (July 2015), 35 pages.
DOI: http://dx.doi.org/10.1145/2791120

1. INTRODUCTION

Modern enterprise applications and systems most often function well but are still known to sometimes exhibit unexpected and unwanted performance behaviours with associated cost implications and failures [Pertet and Narasimhan 2005]. These performance behaviours or anomalies are often the manifestations of bottlenecks in the underlying system. In fact, many factors such as varying application load, application issues (e.g., bugs and updates), architectural features, and hardware failure have been found to be sources of performance degradation in large-scale systems [Cherkasova et al. 2009; Magalhaes and Silva 2010]. Regardless of the sources of the problem, the challenge is how to detect performance anomalies and how to identify potential root-causes. The scale, dynamics, and heterogeneity of today's IT infrastructure further aggravate the problem.

This work is supported by the Swedish Research Council (VR) under contract number C0590801 for the Cloud Control project and the European Union' s Seventh Framework Programme under grant agreement 610711 (CACTOS).
Authors' addresses: Department of Computing Science, Umeå University, Umeå SE-90187, Sweden; emails: {muyi, elmroth, francisco}@cs.umu.se.

DOI: http://dx.doi.org/10.1145/2791120

Performance bottlenecks and anomalies are barriers to achieving predictable performance guarantees in enterprise applications and often come with significant cost implications. Studies [Kissmetrics 2014] have shown that there exist correlations between the end-user performance and sales or number of visitors in popular web applications and how consistently high page latency increases the page abandonment rate. It was also shown that for a small-scale e-commerce application with a daily sales of $100,000, a 1-second page delay could lead to about 7% loss in sales annually. Also according to Huang [2011], Amazon experiences a 1% decrease in sales for an additional 100ms delay in response time while Google reports a 20% drop in traffic due to 500ms delay in response time. These implications show not only the importance but also the potential economical value of robust and automated solutions for detecting performance problems in real time.

Similarly, if left unattended, performance bottlenecks may eventually lead to system failure and outages spanning minutes to weeks. Bottleneck conditions, such as system overload, and resource exhaustion have been reported to cause prolonged and intermittent system downtimes [Pertet and Narasimhan 2005]. Global web services such as Yahoo Mail, Amazon Web Services, Google, LinkedIn, and Facebook have recently suffered from such failures [McHugh 2013]. Unplanned downtimes have significant cost implications [Evolven 2011] not just in lost sales but also in man-hours spent on recovery. To achieve guaranteed service reliability, performance, and Quality of Service (QoS), timely detection of performance issues before they trigger unforeseen service downtime is critical for service providers [Guan et al. 2011]

Considerable efforts have been made to address this issue in the academia with interesting proposals. Many of these solutions leverage the power of statistical and machine learning techniques. Though many of these efforts have been concentrated on solving the problem in specific application domains, the characteristics of the problem and proposed solutions are similar. A basic performance anomaly detection and bottleneck identification (PADBI) system observes, in real time, the performance behaviours of a running system or application, collects vital measurements at discrete time intervals to create baseline models or profiles of typical system behaviours. It continuously observes new measurements for deviations in order to detect expected or unexpected performance anomalies and carry out root-cause analysis to identify associated bottlenecks. This survey aims at providing an overview of the problem and research on the topic. We provide a basic background on the problem with respect to the fundamental elements of the process, methods, and techniques while identifying research trends and open challenges.

1.1. Our Contribution

This work is an attempt to provide thorough description of the performance anomaly detection and bottleneck identification problem and to present the extensive research done in this area. The diverse nature of works addressing this problem informs this work, and we herein present our findings. A similar survey on the general problem of anomaly detection is presented in Chandola et al. [2009]. We start by giving a general background while identifying core elements of the problem. Then we discuss the main contributions of various authors organized in terms of the systems, goals, and techniques used. We conclude by discussing research trends, future directions, and specific requirements for Cloud computing.

1.2. Organization

We introduce the article in Section 1. Section 2 presents a background of the problem and discusses the concept of performance anomalies and bottlenecks, their root-causes, and other fundamental concepts. In Section 3, we address the various detection

strategies and techniques employed in existing literature. Section 4 summarizes past and present research trends while also describing specific Cloud computing requirements in Section 5. Section 6 discusses important concerns about detection methods and presents future directions in terms of challenges and open issues. We conclude in Section 7.

2. BACKGROUND

2.1. Basic Concepts

The performance of computer systems is typically characterized in terms of the duration taken to perform a given set of tasks or the rate at which these tasks are performed with respect to the amount of system resources consumed within a time interval [Gregg 2013].

Performance metrics are key performance indicators (KPI) derived from fundamental system measurements (such as count, duration, and size) to describe the state of operation of a computer system. The two most popular metrics are the *response rime* (or *latency*) and *throughput*. Latency is broadly used to describe the time for any operation to complete, such as an application request, a database query, a file system operation. The throughput of a system is the rate of work performed. For instance, in web applications, throughput is the number of users' requests completed within a time interval (e.g., requests or transactions completed per second) [Gregg 2013].

System *resources* includes physical components such as the CPU, memory, disk, caches, network and virtual components such as the network connections (e.g., sockets), locks, file handles or descriptors [Gregg 2013]. Resource *capacity* describes the storage size or processing strength of a given resource, such as the number of CPUs, and the size of physical memory or disk.

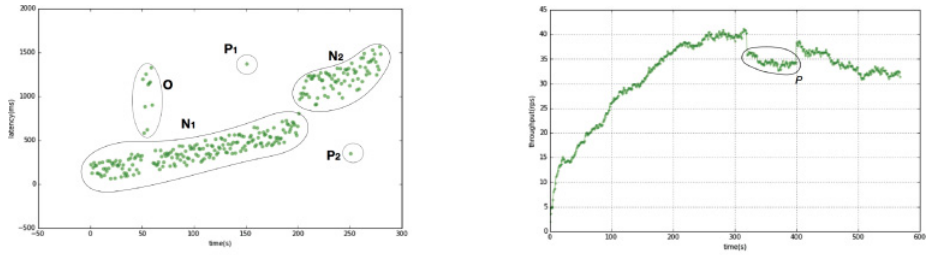
Resource *utilization* of an application typically captures the amount of capacity used with respect to the available capacity. For example, CPU usage is measured as the amount of time (in percentage) the CPU is busy executing instructions from an application, while memory utilization measures (in percentage) amount of storage capacity consumed by a particular process or application. Utilization of network resources may capture the ratio of number of packet transmitted to the full transmission capacity of a network link in a given time interval [Shallahamer 1995; Lilja 2005].

In the following sections, we present various aspects of the PADBI problem.

2.2. Performance Anomalies

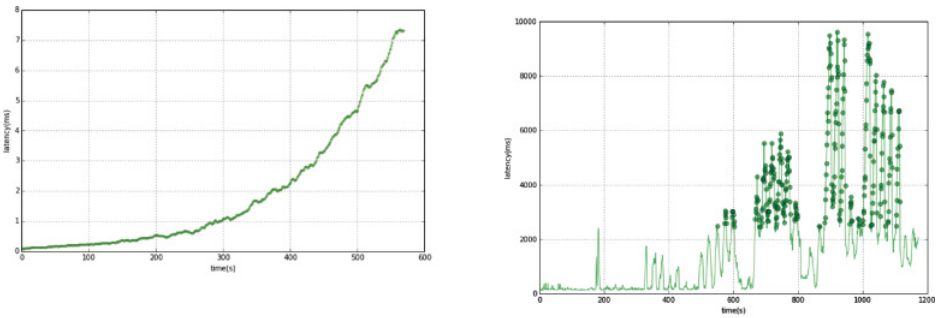
Generally, anomalies can be seen on a graph, as a point or group of data points lying outside an expected normal region [Das 2009]. In performance studies, the data points are discrete measurements of a performance metric, throughput, for example. Figure 1(a) is a plot of *latency* against *time* for an hypothetical system. The two homogeneous clusters (N_1 and N_2) represent the normal operating region, while the points (p_1 and p_2) or group of points (O) falling outside the normal regions are anomalies or outliers. Figure 1(b) captures another example of throughput anomaly, the group of points P represent a short dip in system throughput.

2.2.1. Types of Anomalies. Chandola et al. [2009] identify three basic types of anomalies: *point*, *collective,* and *contextual*. These types only capture anomaly in terms of individual or contiguous data points; however, performance metrics are also known to commonly exhibit characteristic shapes when a resource is saturated [Gunther 2004]. Therefore, we present one more type of anomaly, the *pattern* anomaly, which characterizes performance behaviours in terms of the structure or shapes of their curves rather than finite data points [Gunther 2011].



(a) Latency anomaly. (b) Throughput anomaly.

Fig. 1. Illustration of anomalies.



(a) Anomalous latency growth pattern. (b) Latency spikes.

Fig. 2. Latency anomalies.

(1) *Point anomalies.* A point anomaly is any point that deviates from the range of expected values in a given set of data. For example, a memory usage value 3 standard deviation from the mean (i.e. $>= \mu \pm 3\sigma$) may be considered a point anomaly if the expected behaviour is 1 standard deviation from the mean (i.e., $<= \mu \pm 1\sigma$). In Figure 1(a), points labeled p_1 and p_2 are point anomalies. Point anomalies are the dominant type of anomalies in majority of literature that we reviewed. They commonly manifest as spikes in application latency or system resource utilization measurements. Figure 2(b) shows a plot of application latency with respect to time. The solid dots indicates detected point anomalies.

(2) *Collective anomalies.* Collective anomaly is a homogeneous group of data points deviating from the normal regions of the rest of the data. Though the individual data points may not be anomalous with respect to the group, their occurrence together as a collection is anomalous. An unexpected streak of low-throughput values may be considered anomalous when compared with higher-throughput behaviour in past observation windows. In Figure 1, the group of points labeled O in Figure 1(a) and points labeled P in Figure 1(b) are collective anomalies.

(3) *Contextual anomalies.* Some performance anomalies manifest only under specific execution environments or contexts. The contexts may be defined by load levels (e.g., high, moderate, load, or bursty), type of payloads (e.g., IO-bound, CPU-bound, read-heavy, write-heavy or mixed), system states (e.g., system configurations), or by the nature of underlying computing infrastructure (e.g., virtualized or shared-hosting environments), and so on.

(4) *Pattern anomalies.* The shapes of some performance metrics when plotted are known to exhibit specific pattern that can be used to identify anomalous behaviours

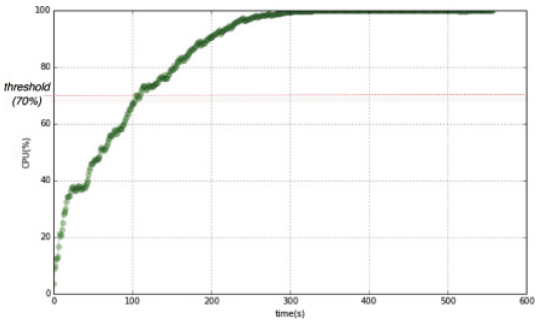


Fig. 3. CPU saturation bottleneck.

[Gunther 2011]. For example, application latency is known to exhibit an asymptotic growth as seen in Figure 2(a). The shape or pattern of performance metric may be anomalous if it does not conform to the shape of typical behaviour. Pattern anomalies may be considered a generalized form of collective anomalies because the anomalous shape is made up of a set of data points that are also collective anomalies.

2.3. Performance Bottlenecks

A bottleneck is a resource or an application component that limits the performance of a system [Gregg 2013]. Malkowski et al. [2009] describe a bottleneck component as a potential root-cause of undesirable performance behaviour caused by a limitation (e.g., saturation) of some major system resources associated with the component [Lee et al. 2012]. Such components often exhibit frequent congestion of load [Mi et al. 2008a]. Also, application or system metrics correlating with an observed performance limitation are referred to as bottleneck metrics [Parekh et al. 2006].

2.3.1. Types of Bottlenecks.

2.3.1.1 Resource Saturation Bottlenecks. A resource is saturated when its capacity is fully utilized or past a set threshold [Gregg 2013]. For example, Figure 3 depicts a saturated CPU past the threshold usage level of 70%. According to Gregg [2013], saturation may also be estimated in terms of the length of a resource queue of jobs or request to be served by that resource. Saturation causes different system resources to be bottlenecked differently with varying performance impact. CPU—near 100% utilization resulting in congested queue and growing latency. Memory—constrained capacity due to limited physical memory or deprivation caused by *memory leaks*[1] leading to constant paging and swapping. Disk Saturation—constant disk access beyond available bandwidth forcing new IO requests to queue up. Network saturation—network congestion due to fully utilized bandwidth causing new traffic to be delayed or dropped.

2.3.1.2. Resource Contention Bottlenecks. In multitasking environments, application processes contend for limited system resources such as CPU cycles, IO bandwidth, and physical memory, and also software resources such as buffers, queues, semaphores, and mutexes. The impact of such contention is well pronounced in cloud data centers due to resource interference between multiple cloud tenants. The *noisy neighbours* effect is an analogy for this interference [Pu et al. 2010]. Several contention scenarios are well known for different system resources: (1) CPU contention—multiplexing the CPU

[1]Memory leak is a classical memory bottleneck scenario where an application indiscriminately allocate memory spaces that are never deallocated thereby saturating the memory and starving other users.

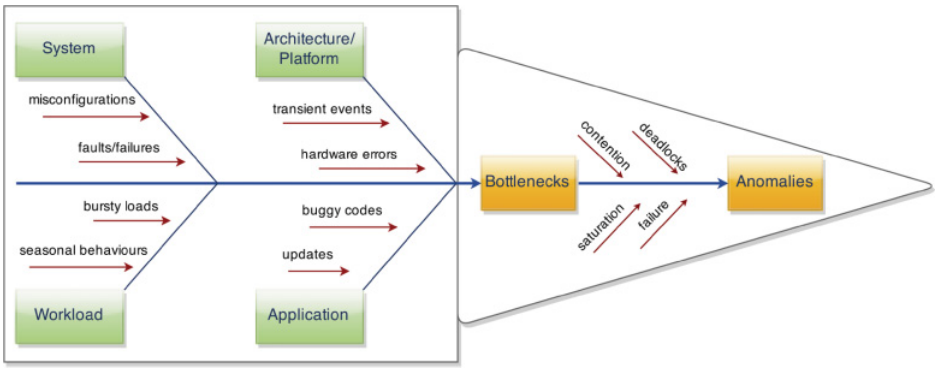


Fig. 4. Cause-effect relationships of performance anomalies and bottlenecks.

between multiple processes causes frequent congested queue and performance interference in virtualized systems especially in the presence of CPU *hogging*[2] programs. Memory contention—sharing limited memory bandwidth and processor-memory interconnect among processes may result in significant performance impact. (3) Disk Contention—the processor-IO performance gap and restricted disk payload[3] causing substantial performance loss especially in IO workloads. (4) Network Contention—excess demands for communication links at peak times lowers the effective bandwidth offered resulting in undesirable network contention delays.[4]

2.3.2. Bottlenecks Behaviours. Performance bottlenecks manifests in different ways depending on applications and systems.

(1) *Single Bottlenecks.* Single bottlenecks exhibit predominant saturation at a single resource or component. An inherent characteristic of the bottleneck component is a near-linear load-dependent growth in resource usage. Malkowski et al. [2009].
(2) *Multiple Bottlenecks.* Two or more system resources or component may saturate simultaneously, or concurrently due to interdependency. Malkowski et al. [2009] classifies multiple saturation behaviours as simultaneous, oscillatory, and concurrent depending on saturation frequency given the presence of another saturation.
(3) *Shifting Bottlenecks.* Shifting bottlenecks are a special case of multiple ones. Due to fluctuating loads and the cascading nature of web requests, an application may experience shifts in bottleneck between two or more application components—the *domino* effects—due to interdependency between them.

2.4. Sources of Performance Anomalies and Bottlenecks

Figure 4 is an extended *Fish-bone* diagram explaining the interrelationships between performance bottlenecks, anomalies and their causes. The green boxes on the left are the main categories of root-causes, the red horizontal arrows are example of primary causes that further explain each category. The orange rectangles on the right are the main effects of the primary causes. Secondary causes that further explain primary causes and effects are represented with red slanted arrows. The thick horizontal arrow,

[2]CPU hogging programs place excessive demand on compute resources thereby impacting the performance of other applications on the same host.
[3]Disk payload is in terms of size (in bytes) and number of IO requests (read/write) per second.
[4]Network contention delay is expressed as ratio of possible demand for a given network link to its maximum capacity.

the *spline*, depicts how primary and secondary causes can be used to explain main effects from left to right.

2.4.1. Application Issues. Application-level issues such as incorrect tuning, buggy codes, and software updates are examples of bottleneck sources [Kelly 2005a]. Incorrect application configuration and updates may introduce unexpected resource bottlenecks [Cherkasova et al. 2009].

2.4.2. Workload. Bursty application loads are characterized by periods of continuous peak arrival rates that significantly deviate from the average or expected workload intensity. Internet phenomena such as *flash-crowds*[5] culminate into workload burstiness [Mi et al. 2008a]. The undesirable effects of such load behaviour include congested queues, oversubscribed threading resources, present of short and uneven peaks in resource and performance measurements [Casale et al. 2009].

2.4.3. Architectures and Platforms. Transient events such as those introduced by underlying system architecture or operating systems (e.g., memory hardware errors), occur over short timespan. Multiple occurrence of such transient errors and events results in bottlenecks that are hard to detect. Wang et al. [2013a] and Mi et al. [2008a] have shown how JVM garbage collection and Intel SpeedStep technology can induce bottlenecks. In modern systems with multicore NUMA architecture, the location of memory relative to a processor may affect application performance especially those with memory-bound workloads [Panourgias 2011].

2.4.4. System Faults. Faults in system resources and components may considerably affect application performance [Muppala et al. 1991] with significant cost. System failures may be intermittent, transient. or even permanent. Reasons for such failures can be attributed to software bugs, operator error, hardware faults, environmental issues, and security violations. In recent times, many popular web application services have been hit by failures that temporarily disrupt their application services for some time [Pertet and Narasimhan 2005].

2.5. Core Elements of the Problem

2.5.1. Nature of the Problem. The complexity of today's systems makes the process of detecting performance issues and identifying root-causes nontrivial. We identify the following as the major challenges:

(1) *Dynamic Dependency*. At scale, applications comprise of multiple interdependent components deployed in data centers servers with heterogeneous and equally interdependent resources. This dependency results in dynamic behaviours. For example, hard-to-detect alternating or cascading bottlenecks between two or more components and resources is very common in large datacenters [Wang et al. 2013b].
(2) *Dynamic Anomaly Characteristics*. Today's systems are by nature highly dynamic with characteristic unpredictable behaviours. It follows that defining a priori all possible behaviours (normal or anomalous) of an application is technically unrealistic. Similarly, the notion of normality, anomalies and their characteristics vary widely across applications, execution environments, and load contexts. Therefore, it is hard to precisely distinguish normal application behaviours from anomalous behaviours at runtime [Lan et al. 2010].
(3) *Nature of Data*. There exist a diverse set of data collection tools, each generating output data in different formats and semantics. This makes it difficult to consume

[5]A flash-crowd is an Internet phenomenon where a network suddenly receives a huge influx of traffic due to breaking news, major events, natural disasters, and so on.

Table I. Recent Literature by Goals

PAD	PBI	PADBI
Zhang et al. [2007b]	Chen et al. [2002]	Cohen et al. [2004]
Gunter et al. [2007]	Malkowski et al. [2007]	Kelly [2005a]
Yang et al. [2007]	Chung et al. [2008]	Jung et al. [2006]
Cherkasova et al. [2008]	Ben-Yehuda et al. [2009]	Agarwala et al. [2007]
Lan et al. [2010]	Iqbal et al. [2010]	Kandula et al. [2009]
Fu [2011]	Xiong et al. [2013]	Malkowski et al. [2009]
Sambasivan et al. [2011]	Wang et al. [2013b]	Magalhães and Silva [2011]
Tan et al. [2012]		Magalhaes and Moura Silva [2011]
Pannu et al. [2012]		Lee et al. [2012]
Wang et al. [2012]		Kang et al. [2012]
Guan and Fu [2013b]		Dean et al. [2014]
Yu and Lan [2013]		
Sharma et al. [2013]		
Huang et al. [2013]		
Wang et al. [2014]		

the data in a uniform manner [Lan et al. 2010]. Also, due to the influence of varying data collection mechanisms, processing and transmission errors, performance data may suffer from the presence of noise whose values may be similar to anomalies. This complicates the detection problem, as noise often masquerades as anomalies resulting in high false-positive detections.

Furthermore, today's systems generate huge quantity of health or operational data that can easily overwhelm analysis and detection process. Finding anomalies and bottleneck symptoms in such datasets is analogous to finding a needle in a haystack.

2.5.2. System Goals. The general research questions behind PADBI systems are:

(1) *How to automatically detect anomalous performance behaviours?*
(2) *How to automatically identify the root cause of an observed performance anomaly?*
(3) *Which system resource or component is responsible for an observed violation of a performance objective?*

We refer to the goal of systems addressing the first question as *performance anomaly detection* (PAD). Systems addressing the second and third question are classified as *performance bottleneck identification* (PBI). In many cases, we observed a blurred line between papers addressing anomaly detection and those addressing bottleneck detection. We categorize such systems (i.e., addressing all the three questions) as *performance anomaly detection and bottleneck identification* (PADBI).

Table I classifies recent literature according to their goals, with PAD, PBI and PADBI systems accounting for 53%, 29% and 18% respectively of all literature reviewed as presented in Table IX and X of Appendix A.

Generally, the output of PADBI systems may include a set of anomalous performance *indices*, a *timestamp* (time of incident), a set of *anomalous metrics*, a *label* (in case of learning based systems)—an assigned class to which a sample belongs e.g. *normal* or *anomaly*, and an *anomaly score*—the degree to which a case is considered anomalous. The performance of PADBI systems themselves and their sensitivity are evaluated based on the following metrics:

(1) *Precision.* This is the ratio of correctly detected anomaly to the sum of correctly and incorrectly detected anomalies. It is also referred to as the *positive predictive value* (PPV) in literature.

(2) *Recall.* Also known as the *confidence score* or *true positive* (TP) rate. Recall is the ratio of correctly detected anomalies to all anomalous instances in a given dataset. It may be referred to as *densitivity* in some literatures. Conversely, the ratio of correctly detected normal instances to the total count of normal instances in the dataset is called *specificity* or the *true negative* (TN) rate.
(3) *Accuracy.* This is the ratio of the count of all correct detections (anomalies or not) to the total number of cases in the system.

2.5.3. Systems. PADBI have been studied in many system domains and application architectures. These include distributed applications (such as web-based client-server and multitier application) deployed in dedicated and shared server environments. Distributed applications are composed of highly specialized application entities integrated to achieve some high-level system objectives [Peiris et al. 2014]. While the Grid [Yang et al. 2007] is known for running short- and long-term applications performing large computations across distributed nodes, cloud infrastructures [Gong et al. 2010] allows diverse applications to share virtualized system resources (storage, compute, and network). The complexity of a system can be estimated by the number of resources and the composition of its applications.

System Resources. Resource demands are essential indicators of performance problems. The number of resources determines the size of the metric space and the volume of data that are eventually gathered. The interdependencies between system resources enables faults to propagate the system in a cascade manner (e.g. Disk and CPU resources).

Application Components. Modern applications are composed of heterogeneous software components distributed across separate and often geographically dispersed physical servers. In virtualized environments, application components are deployed in virtual machines (VMs) that can be migrated from one physical node to another within and across data centers. This complex composition and deployment brings special requirements for localization of performance problems.

Table II is a classification of research contributions according to system and application domains addressed.

2.5.4. Data. Performance data are a time series of the values of a set of performance metrics, systematically sampled over a regular interval. In this section, we briefly outline important aspects of such data.

Characteristics of Performance Data. Performance data are quantitative in nature. A performance metric is an attribute of a system or its component parts defining a state of the system. In general terms, metrics may also be referred to as *features* in some literature. A *case* or *instance* is a set of closely related features—a vector capturing a particular state of system at a point in time.

Sources of Performance Data. The bulk of performance data come from extensive measurements of metrics at two levels. *Application metrics* are foreground or in-band metrics that captures the current state or health of an application. Examples include application *response time* and *throughput*, number of *application users,* and *database connections*. *System metrics* are background or out-of-band metrics that capture the current state of the underlying system. Background metrics encompass not only resource utilization metrics but also hardware counters and error events. Examples are *CPU utilization*, number of *IO read*/*write* requests, *IO wait time*, and *CPU queue length*.

2.5.5. Data Collection. Monitoring is used to observe the runtime performance of a system by collecting both application- and system-level metrics using automated

Table II. Recent Literature on Systems

Reference	Dedicated Server	Virtualized, Cloud	Grid	Distributed	Web	Multitier
Wang et al. [2014]				✓	✓	✓
Dean et al. [2014]		✓				
Peiris et al. [2014]				✓		
Xiong et al. [2013]		✓		✓	✓	✓
Guan and Fu [2013a]		✓		✓		
Wang et al. [2013]		✓			✓	
Yu and Lan [2013]				✓		
Nguyen et al. [2013]		✓		✓	✓	✓
Sharma et al. [2013]		✓		✓	✓	✓
Tan and Adviser-Gu [2012]		✓		✓	✓	✓
Dean et al. [2012]		✓		✓		
Casale et al. [2012]		✓				✓
Fu et al. [2012]		✓		✓	✓	
Rathfelder et al. [2012]	✓			✓	✓	
Kang et al. [2012]		✓		✓		✓
Magalhaes and Moura Silva [2011]	✓			✓	✓	✓
Yu et al. [2011]		✓		✓	✓	✓
Do et al. [2011]		✓		✓	✓	✓
Lan et al. [2010]	✓					
Ben-Yehuda et al. [2009]		✓			✓	
Kandula et al. [2009]		✓		✓		
Gu and Wang [2009]	✓			✓		
Mi et al. [2008a]		✓		✓		✓
Gunter et al. [2007]			✓	✓		
Yang et al. [2007]			✓	✓		
Malkowski et al. [2007]	✓			✓	✓	✓
Agarwala et al. [2007]	✓			✓	✓	✓

third-party tools or via built-in Kernel counters. The efficiency of the detection process is influenced by three major aspects of data collection that are discussed next.

System Observability: White, gray or black box? The observability property of an application is greatly dependent on the type of infrastructure. In dedicated cluster environment, administrators have access to both application source codes and underlying infrastructure (*white-box*), such that both profiling and deep source tracing are possible possible. Whereas in cloud environments, cloud providers see applications as *black-boxes* while service providers (application owners) lack a global view of the infrastructure outside of their VMs. In general, white-box and gray-box systems allows for full and partial source code instrumentation, respectively. Such modifications are generally intrusive with significant runtime overhead. Black-box applications expose detail visibility into the application, thus limiting the amount of insights achievable but they are profiled in a nonintrusive manner [Nguyen et al. 2013].

Profiling vs. Tracing. Profiling extends beyond logging the state of system to studying the resource consumption behaviour and dependencies in order to assess the overall performance of the system. It also involves establishing analytical models that may be used to describe the dynamics of the system and predict performance [Shende 1999]. Examples of popular profiling tools include ps, *sysstat*, *htop*, *top*, *collectd*, *Nagios*, *Ganglia*, *apachetop*, *dstat*, and *iftop*.

Tracing is used to track fine-grained network or source-level events and misbehaviour via source code instrumentation. In addition to tracking the occurrence of certain events, tracing may reveal the execution flow, actions performed, caller-thread, and time spent in specific code blocks [Passing 2005]. Runtime code instrumentation is a common tracing method. Tracing platforms such as Aspect Oriented Programming [Tarby et al. 2007] and Java Byte Code instrumentation [Lee and Zorn 1997; Binder et al. 2007] have been used to observe applications. Examples of third-party tracing tools are *KProbe*, *AspectJ*, *JimysProbe*, *Dtrace*, *Magpie*, and *Strace*.

Influence of sampling interval. The volume of data generated by monitoring depends not only on metric space but also the rate at which we collect them. Shorter sampling intervals give finer resolutions than longer ones with additional compute and storage overheads. Longer sampling intervals produce lighter data but may miss out on transient performance events. An adaptive and selective monitoring is proposed in [Magalhaes and Moura Silva 2011]. The technique begins with a baseline sampling interval and continuously adjust the interval on the fly to adapt to the changing application behaviour. Also, metrics may be sampled selectively on demand.

3. SOLUTION STRATEGIES AND METHODS

Conventionally, the approach to detecting performance problems involves continuous estimation of models of normal system behaviours at specific points of interest. New performance observations that fail to match (within some acceptable confidence levels) existing models are flagged anomalous and system administrators alerted accordingly [Sharma et al. 2013]. However, many solutions employ more complicated techniques (such as statistical and learning methods) while following one or more strategies to achieve some detection goals. Different detection strategies and techniques are presented in Sections 3.1 and 3.2, respectively.

3.1. Detection Strategies

Existing PADBI systems often follow one or more strategies for robustness. A strategy defines the set of policies to achieve a detection goal. The choice of strategy is greatly influenced by system observability as well as whether the detection is to take place in either *offline* or *online* mode. Offline detection is a "post-mortem" identification and analysis of performance issues. Online detection is performed at run-time.

All strategies use thresholding to prune and complement detection decisions in one way or the other. A threshold is a limit value or range of values for parameters or metrics of interest beyond which an event is raised. Example thresholds include the p-value and R^2 (coefficient of determination) in statistical detection, distance from centroid (clustering), and entropy bounds (information theoretic) in machine learning detection. Setting thresholds becomes cumbersome when many parameters and metrics are involved. Modern system exhibit dynamic behaviours that consistently violates the ideal set thresholds. It is therefore expedient that the right thresholds are estimated. Threshold values are also expected to evolve with respect to change in underlying execution environment. In addition, it is crucial to understand the sensitivity of varying thresholds on detection accuracy.

Based on existing literature, we have identified four important strategies presented in Sections 3.1.1 through and 3.1.4. References of research contributions in each category can be found in Tables IX and X of Appendix A.

3.1.1. Signature-Based Detection. Applications exhibit specific behaviours at runtime that characterizes their performance such as their resources utilization, their performance, and load saturation rates. Such runtime characteristics are called *signatures*, *fingerprints,* or *profiles*. PADBI systems generate signatures as compact runtime

representations of important performance behaviours of an application. A signature may capture a normal system state or a deviation from that anomaly signature. Signature-based detection is a data-driven approach that consumes output of application profiling or tracing. Baseline signatures of prevailing system behaviour are generated in real-time and are then used to filter new observations for unwanted behaviours. An important attribute of signature-based detection is that signatures are discovered at runtime and may not require that a signature history is maintained. Generally, signature-based strategies require domain knowledge of and global snapshot of system state to achieve high accuracy. They usually record low false-positive detection and are suitable for known anomalies [Bodik et al. 2010; Bodík et al. 2008; Mi et al. 2008b; Cohen et al. 2005; Cherkasova et al. 2009].

3.1.2. Observational Detection. Applications can be observed through direct experimentation, staging (usually in a controlled environment) followed by in-depth analysis of observed anomalies and root-cause identification. To collect data, applications are either profiled in a black-box manner or source code instrumented for tracing. This approach covers both real-time analysis and "post-mortem" analysis of log files to discover sources of problems. Observational detection may also involve the staging of applications and systems where faults, anomalies, and bottlenecks are deliberately injected in order to understand system behaviour under such conditions. This approach is beneficial in various ways. First, it helps to prevent the limitation of hasty assumptions found in systems based only on analytical and simulation models. Second, it enables the understanding and identification of intrinsic behaviours of a system. Because this approach depends mostly on experiential and cognitive knowledge, it yields high accuracy in detecting known and unknown anomalies. This however, makes it difficult to implement in real-time situations because the experiential knowledge of right thresholds, transient anomaly behaviours have to be encoded into an automatic mechanism [Pu et al. 2007; Magalhaes and Moura Silva 2011; Tan et al. 2010].

3.1.3. Knowledge-Driven Detection. In specific enterprise systems, performance issues are often periodic with known root-causes and potential remedies. Many research and industrial systems leverage on such known anomalies and bottleneck definitions to identify and address performance problems. Knowledge-based detection approach identifies performance issues and their causes based on historical records of previously observed anomalies. It maintains a dynamic store (knowledge base) where definitions of known anomalies, their possible root-causes are maintained. The detection of new issues often trigger an update of the knowledge base. These definitions are converted into a set of formal rules that can be manipulated by an inference engine to detect performance issues and identify the root-causes. Although there exists some similarity between knowledge-based and signature-based detections, generation of rules and definitions does not necessarily have to be entirely at run-time in the former. This contrasts the online generation of signatures in the latter and does not require specialized inference engines. Knowledge-based detection is typically a data-driven approach and also require a great deal of understanding of the application and system domains. This strategy have high true-positive detection of known performance issues [Chung et al. 2008; Koehler et al. 2011; Li and Malony 2006].

3.1.4. Flow and Dependency Analysis. By studying the flow of communication across components in distributed applications, performance anomalies and hotspots can be easily identified. This approach typically involves real-time collection and analysis of traffic data (such as SNMP and TCP packets). In black-box systems, in-bound and out-bound network traffic may be observed to understand the performance behaviour of specific application components. Similarly, in white- or gray-box environments, dynamic code

tracing may be used to trace application requests or method invocations across code segments or network boundaries to better understand performance issues, their contexts and to pinpoint their sources. To detect anomalies and their causes, frequency, correlation, and causal path analysis are usually performed. Of great concern is the potential data collection overhead involved in this approach especially in large-scale systems with hundreds of applications and components. This approach often yield fine-grained detection with high true positives in black-box environments. Conversely, observing and understanding in-out traffic of hundreds of black-box components without prior knowledge may yield high false-positive detections [Ben-Yehuda et al. 2009; Sambasivan et al. 2011; Agarwala et al. 2007; Aguilera et al. 2003; Nguyen et al. 2013].

3.2. Detection Methods

To detect performance anomalies and identify associated bottlenecks, methods from diverse fields have been used, prominently from the domain of statistical analysis, machine learning (ML). We focus our discussion on statistical and learning techniques due to the volume of literature on them. However, signal processing methods such as *Extended Window Averaging*, *Adaptive Filtering,* and *Fourier Transforms* have also been used in Yang et al. [2007] and Malkowski et al. [2009]. We describe the commonly used techniques in literatures along with relevant references in Sections 3.2.1 and 3.2.2.

3.2.1. Statistical Detection. Statistical techniques provide capabilities to detect trends or drifts in critical performance metrics. Typically, researchers and system administrators observe system behaviours over time to make sense of underlying system dynamics. They construct models to hypothesize their observations, and employ some methods to estimate key model parameters and the relationship between them. Many statistical methods assume that some characteristics of the data are known a priori or can be inferred. For example, assuming the probability density of a performance metric follows a Gaussian (normal) distribution. These are called *parametric* statistical techniques. Examples of such methods are *Tukey limits*, *ANOVA* tests, *Pearson correlation*, *Grubb's Maximum normed residual,* and the *Student-t* tests. Nonparametric methods also exist that require little or no assumptions about the underlying nature of the data. Instead of assuming distribution of data as Gaussian, methods such *Histogram* or *Kernel* functions are used to estimate data distributions [Chandola et al. 2009; Rajasegarar et al. 2008]. The *Median*, *CUSUM*, *Spearman correlation*, *Kruskal-Wallis,* and *Wilcoxon's* tests are examples of nonparametric statistics [Burke 2001].

In general, statistical analysis provide a strong theoretical basis for detecting, and quantifying the influence of anomalies and bottlenecks on system performance. The assumption that the distribution of data is known a priori in many cases qualifies them for identifying well-known anomalies. However, many statistical methods exhibit sensitivity to variation especially when assumptions about the distribution of the data do not hold.

3.2.1.1. Gaussian-Based Detection. Gaussian-based techniques generally exploit the assumption that underlying data distribution is normal. Such techniques build Gaussian models parameterized by the mean μ, and variance σ^2 (i.e., $X \sim N(\mu, \sigma^2)$) [Chandola et al. 2009; Markou and Singh 2003].

The Tukey [1977] limits detect anomalous data points based on the distance from the distribution mean. The lower and upper normal thresholds are set at $(Q_1 - k * IQR)$ and $(Q_3 + k * IQR)$, respectively, where Q_1, Q_3 and IQR (computed as $Q_3 - Q_1$) are the 1st quantile, 3rd quantile, and the interquantile range, respectively. Data points outside this range are flagged anomalous. Though the threshold limit k is by default

1.5, it can be set to an appropriately chosen scalar for specific application [Wang et al. 2011].

The density distribution may also be exploited for detecting anomalous data points based on the Gaussian Mixture Model (GMM) [Markou and Singh 2003]. GMMs are parametric models of the probability distribution of continuous random variables estimated using the iterative Expectation-Maximization (EM) algorithm [Reynolds 2009].

Given a dataset X composed of n normally distributed features $\{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$ with corresponding parameters $\{\mu_1, \mu_2, \ldots, \mu_n\}$ and $\{\sigma_1^2, \sigma_2^2, \ldots, \sigma_n^2\}$, the Gaussian probability density function (PDF) for each feature $x^{(i)}$ is defined as $P(x^{(i)}; \mu_i, \sigma_i^2) = \frac{1}{\sigma_i\sqrt{2\pi}} \exp(-\frac{1}{2}(\frac{x^{(i)}-\mu_i}{\sigma_i})^2)$. The detection procedure proceeds first by estimating parameters μ_i and σ_i for each feature x_i. A new observation of the form $X = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$ is classified anomalous if $P(X) < \epsilon$, where $P(X)$ is the sample's probability of being normal. The combined PDF of the dataset $P(X)$, is estimated as the product of the PDFs of each feature, that is, $P(X) = \prod_{i=1}^{n} P(x^{(i)}; \mu_i, {\sigma_i}^2)$. The value of ϵ can be varied depending on application requirements [Hodge and Austin 2004; Walck 2007].

Other density-based methods include the *Parzen* Windows Estimation [Parzen 1962], the *Grubb's* test [Grubbs 1969] and the *Student's t*-test [Markou and Singh 2003].

3.2.1.2. Regression Analysis. Regression analysis is a methodology for investigating relationships between performance metrics and to quantify the statistical significance of such relationships. For instance, regression analysis may explain how variation in load influences a given KPI with the assumption that the relationship (linear or nonlinear) between them is known a priori. The goal of regression is to estimate the set of model parameters that minimizes the absolute or the squared error. Commonly used algorithms for estimating model parameters include *Ordinary Least Squares (OLS)*, *Least Angle (LA)* and *Recursive Least Square (RLS)* [Kleinbaum et al. 2013].

For example, given a linear model of the form $T = \alpha(U_{cpu})$ describing the relationship between throughput and CPU utilization, a regression-based PADBI system first fits this model on a training data to estimate parameter α, the standardized p value, and the coefficient of determination (R^2). And for each test instance, the model computes the residuals—the variability in the test instance not explained by the model. The magnitude of the residuals are used to determine an anomaly score [Chandola et al. 2009]. New observations falling outside the confidence interval produced by the model may be classified as anomalous [Courtois and Woodside 2000; Lee and Brooks 2006]. An interesting use of regression models in modeling enterprise web applications is in generating *Transaction Mix* (TM) models. These models are used to describe application performance as a function of the mix of transactions (or requests) processed per unit time and their corresponding resource utilization. They are generally used for capacity planning and detecting transaction performance problems. Table III outlines literatures based on regression models.

3.2.1.3. Correlation Analysis. Correlation quantifies the degree of association between performance metrics. The interdependency of variables are estimated as a coefficient R in the range -1 to $+1$. Positive R values indicates a trend of increase in one variable as the other increases. Negative R values is a trend of decrease in one variable as the other increases. Variables sharing no association have R values of 0.0. Commonly used algorithms to estimate R are; the *Pearson*, the *Kendal* rank and the *Spearman* correlation [Magalhaes and Silva 2010]. Lets look at a simple example of how correlation may be used for detecting anomaly behaviour. Assume the correlation coefficient between two performance metrics A and B have been estimated in the confidence interval $[R_{min}, R_{max}]$ based on some training datasets. New observations of A and B over a time

Table III. Literature on Regression-Based Approaches

Author	Methodology
Kelly [2005a]	Presents a Regression-based TM model for identifying performance anomalies in geographically distributed applications. Tested with commercial applications such as ACME, FT, and VDR.
Zhang et al. [2007a]	Proposes an approach using Nonnegative Least Square (NLS) regressive TM models for estimating resource demands by different client transactions and applicability in resource provisioning.
Cherkasova et al. [2008]	Presents signature-based anomaly detection built on Zhang et al. [2007a]
Cherkasova et al. [2009]	Demonstrates how stepwise linear regression addresses model "overfitting" problem in Zhang et al. [2007a] and further present a segmentation-based method (as an extension of Cherkasova et al. [2008]) for detecting performance changes in enterprise web applications.
Kang et al. [2012]	Present a Regression-based diagnostic framework for analyzing performance anomalies and potential causes of SLA violations in virtualized systems. Their approach is based on Lassio, a variant of the Least Angle Regression (LAR) algorithm to identify suspicious system metrics accounting for observed performance anomaly.
Yang et al. [2007]	Models the relationship between application metrics and system metrics for metric selection, reduction, and anomaly detection.

window $[t_1, t_n]$ in the form $W_a = \{a_1, a_2, \ldots, a_n\}$ and $W_b = \{b_1, b_2, \ldots, b_n\}$ are anomalous if the R value between W_a and W_b falls outside the expected range $[R_{min}, R_{max}]$.

Canonical correlation is an advanced method that has been demonstrated for finding the linear association between one or more performance metrics. Given a vector of metrics $X = \{x_1, x_2, x_3, \ldots x_n\}$ and $Y = \{y_1, y_2, y_3, \ldots y_n\}$, the method computes the canonical variates (u, and v), the orthogonal linear combination of X and Y, that best capture the variability within and between X and Y. Kernel canonical correlation [Huang et al. 2006] is a popular variant of this method. See Table IV for references relating to correlation-based systems.

3.2.1.4. Statistical Process Control. Statistical process control (SPC) [Oakland 2008], is a quality control method widely used to monitor production processes for early detection of undesirable variation in process output. SPC provides a set of control charts, such as CUSUM, Shewart (ImR or XmR) charts, for monitoring process stability and variation. According to Bereznay and Permanente [2006], SPC is not suitable for interval based sampling data such as system performance traces. This motivates the development of the *Multivariate Adaptive Statistical Filtering (MASF)* method. MASF, Buzen and Shum [1995] is a SPC framework for detecting changes in a Gaussian distribution. MASF uses parameters mean (μ), standard deviation (σ), and variance (σ^2) of data collected during normal system operations as the basis for filtering subsequent system measurements for anomalous behaviours. For example, a MASF-based detection policy may set a control limit (CL) at the mean μ, a upper control limit (UCL) at $(\mu + 3\sigma)$ and a lower control limit (LCL) at $(\mu - 3\sigma)$. These control limits describe the range of expected variability in the data over a period of time. When new observations fall outside outside the set control limits, they are detected as anomalies and their cause(s) must be identified and corrected [Wang et al. 2011].

3.2.1.5. Statistical Intervention Analysis. Statistical Intervention Analysis (SIA) [Box and Tiao 1975] measures the form and magnitude of shifts in timeseries data. The shift is considered a consequence of a change, an *intervention* or *shock* in the data. It is particularly useful for studying the impact of an interventions (e.g., change in policy, natural disaster, or breaking news reports) on the behaviours of physical systems. In

Table IV. Literature on Correlation-Based Approaches

Author	Methodology
Agarwala et al. [2007]	Presents a performance management system where correlation analysis is used to identify important performance metrics and estimate the influence of specific application services and system resources on such them.
Magalhaes and Silva [2010] and Magalhaes and Moura Silva [2011]	Proposes an approach to identify potential root-causes of observed performance variation as either due to workload change or application update by computing the Pearson coefficient of correlation between aggregated workload, latency, and system metrics over some time window.
Kang et al. [2012]	Presents a method using correlation analysis to selecting model parameters. By filtering metrics showing collinearity relationships above a set threshold, they are able to reduce the dimension of models for detecting performance anomalies in virtualized infrastructures.
Sharma et al. [2013]	Uses correlation analysis to identify variations in performance metrics in a cluster of Virtual Machines (VM). They also show a technique to characterize anomalies by defining anomaly signatures in terms of changes to correlation between VMs in the cluster.
Wang et al. [2013] and Do et al. [2011]	Presents a Kernel-based Canonical correlation method to discover the correlation between workloads and performance in Internetware and how this is used for anomaly detection.
Gambi and Toffetti [2012]	Presents a novel Kriging-based model of system performance as a function of dynamic resource allocation and workloads to predict and detect performance problems.
Peiris et al. [2014]	Proposes a correlation-based method for automatic identification of associations among performance counters in a distributed system and how the association is used for detecting anomalies.

Table V. Literature on other Statistical Methods

Method	Highlight	Reference
ANOVA	Root-cause identification, and Change detection	Magalhães and Silva [2011] and Bereznay and Permanente [2006]
Index of Dispersion	Workload burstiness and variability detection in web applications	Mi et al. [2008a] and Casale et al. [2012]
Mean Standard Deviation, Cummulative Density Function	Anomaly detection in Grid application	Gunter et al. [2007]
Student's t-test	Localization of anomalous metrics	Wang et al. [2014]
Markov Model	Prediction of anomalous performance metrics and fault localization	Nguyen et al. [2013], Tan et al. [2012], Tan and Gu [2010], and Gu and Wang [2009]
Kernel Density Estimation	Estimation density functions using Kernel regression for inferring resource saturation	Malkowski et al. [2009]
Probability Models	Anomaly prediction and detection	Cohen et al. [2004], Zhang et al. [2005], Kandula et al. [2009], and Tan et al. [2010]
Statistical Process Control	Detection of performance transient and persistent anomalies and identification of anomalous correlation in database and enterprise systems	Trubin and Merritt [2004], Trubin [2005], Brey and Sironi [1990], Lee et al. [2012], Wang et al. [2013], and Bereznay and Permanente [2006]
Statistical Intervention Analysis	Bottleneck identification	Malkowski et al. [2007]

Internet applications, interventions are similar to phenomenal such as Internet *flash-crowds*, the *slashdot* effect, or a node failure.

3.2.2. Machine Learning Detection. Learning algorithms sift through massive metric space to identify patterns of interests or indistinct relationships [Rogers and Girolami 2011]. In performance studies, these patterns may be unexpected behaviours or symptoms of unplanned failures. Machine learning algorithms can be classified into two broad categories based on the nature of input and expected output of the algorithms [Alpaydin 2014].

Supervised Learning. Supervised learning algorithms require well-labeled datasets. Each data instance in a training dataset is assumed to belong to one of several classes (e.g., *normal* or *anomaly*). The goal is to build a generalized model that the captures the relationship between the feature set and each class during the training phase. These models are later used to classify new test instances during the testing phase. The need for well-labeled training data greatly limit the scope of their application for real-time use. They are, however, well suited for recognizing well-known anomalies. The use of supervised techniques in dynamic environments such as cloud data centers is hampered by the cost of retraining due to dynamic reconfiguration of application components and change in underlying execution environments. Supervised learning techniques do not easily lend themselves to frequent updates of training the dataset.

Unsupervised Learning. Unsupervised learning algorithms require no training data and no labeled data. The objective is to discover hidden patterns or regularities in the data, similar to density estimation in statistical. Unsupervised learning techniques cluster input data into classes based solely on their statistical properties. No assumption, however, is made of the distribution of the underlying data. For improved accuracy, it expected that normal data instances are more frequent in the dataset than abnormal instances. Techniques in this category are amenable to changes in the underlying system environment because no training is involved. And are particularly suitable for detecting unknown anomalies in cloud data centers where precise definition of anomaly characteristics may not always exist.

Semisupervised Learning. An emerging approach is to maximize the best of supervised and unsupervised learning. Semisupervised algorithms assume a small chunk of the dataset is labeled usually the normal class and the remaining unlabeled instances are anomalous. They often outperform their supervised and unsupervised counterparts as they leverage the presence of labeled data to identify inherent structure in the data. A similar approach is called the *weakly-supervised* or *bootstrapping* method. This method begins by training the classifier with a few training examples. When the classifiers finds positive test instances, it augments the original training data with the new instance and retrains the classifier. The performance of bootstrapping improves as the size of training data grows given false-positive detection is minimal. Bootstrapping is well suited for large-scale infrastructure where definitions of normality and abnormality evolve according to changing execution context.

The operation of a learning based PADBI system is often enhanced by two preprocessing tasks.

Dimensionality Reduction. To handle problems with many performance metrics, the metric space may be reduced by projecting the metrics to a new space where only the most relevant is preserved. Principal Component Analysis (PCA) is common

Table VI. Supervised PADBI Systems

Reference	Technique	Methodology
Tan and Adviser-Gu [2012] and Tan et al. [2010]	Bayesian classifier and Tree augmented networks (TAN)	Presents a method for predicting and classifying anomalies.
Fu [2011]	Decision trees and TAN	Presents a technique for reducing metric dimensions based on Mutual Information and PCA and identifying performance anomalies Tree-based classifier.
Jung et al. [2006]	Decision tree	Presents a decision-tree based automated approach for detecting performance bottlenecks.
Cohen et al. [2004]	Tree augmented Bayesian networks (TAN)	Proposes a method exploring TANs as a basis for detecting SLO violations and identifying sets of system metrics that caused the violation in multi-tier web applications.
Gu and Wang [2009]	Bayesian classification	A stream-based anomaly detection method is used to detect anomaly symptoms and infer their root-causes.
Parekh et al. [2006]	TAN, Bayesian networks, LogicBoost, C4.5 decision tree.	Explores the performance of various machine learning classifiers with regards to bottleneck detection in an enterprise applications.
Powers et al. [2005]	Bayesian classifiers, Auto-regressive models, Multivariate regression	Presents a comparative study of the performance of three machine learning and statistical methods to predict the number of performance SLA violations.

method for doing this. PCA takes k correlated metrics as input and reduces them to $m \leq k$ nondependent metrics. These m metrics can be interpreted as linear combinations of the original set [Guan et al. 2012; Fu 2011]. Other methods for dimension reduction include factor analysis, independent component analysis, and nonlinear PCA [Fodor 2002].

Similarity Identification. Metrics with high similarity affects the efficiency of learning algorithms such as clustering. A common method of evaluating similarities between features is based on the *Mutual Information* algorithm from the domain of Information Theory [Steuer et al. 2002; Battiti 1994].

Unlike statistical detection, learning techniques do not make assumptions about the underlying distribution of data. We identify a few references of each type of learning in Tables VI, VII, and VIII.

We further describe commonly used learning techniques in literature in Sections 3.2.2.1 through 3.2.2.4.

3.2.2.1. Classification-based Techniques. Classification-based learning algorithms are special cases of supervised learning. The objective is to determine if data instances in a given feature space belongs to one class or multiple classes. During a training phase, the algorithm identify classes and learn a model that associate each class label with the characteristics of features present in the data. The testing phase use these models to classify new data samples. Ruled-based detection systems are a specific example of how classification learning can be used in detecting anomalous behaviours. Common classification techniques include *Decision Trees*, *Support Vector Machines*, *Artificial Neural Networks*, and *Bayesian Networks* [Kotsiantis 2007].

Rule-based techniques. The goal is to learn as many rules that captures normal behaviours of a system as possible. First they discover rules from the training data

Table VII. Unsupervised PADBI Systems

Reference	Technique	Methodology
Wang et al. [2012, 2014]	Local Outlier Factor (LOF)	Proposes an online anomaly detection approach for web applications present an incremental clustering algorithm for training workload patterns online, and employ LOF in the recognized workload pattern to detect anomalies.
Dean et al. [2012]	Self-Organizing Maps (SOM)	Presents an anomaly detection mechanism in IaaS Cloud using SOMs to learn emergent system behaviour and predict unknown anomalies.
Guan et al. [2012]	Bayesian ensemble models	Proposes an hybrid learning approach by characterizing normal execution states of the system as an ensemble of unsupervised Bayesian models and uses decision tree to predict and detect system failures in a Cloud environment.
Huang et al. [2013]	Local Outlier Factor (LOF)	Presents an adaptive method extending the Local Outlier Factor algorithm for detecting both contextual and unknown anomalies in a Cloud system.
Yu and Lan [2013]	Nonparametric clustering	Proposes a decentralized approach for detecting anomalies in Hadoop clusters based on Hierarchical Grouping and majority voting.

Table VIII. Semisupervised PADBI Systems

Reference	Technique	Methodology
Lan et al. [2010]	Principal and Independent Component Analysis	Presents an automated anomaly detection mechanisms for identifying system nodes whose behaviours are deviating from others in a cloud data center.
Pannu et al. [2012]	Classification, Clustering, Support Vector Machines	Presents a self-evolving mechanism for predicting and detecting of system failures in Cloud systems.
Smith et al. [2010]	Bayesian Networks, Principal Component Analysis, Clustering	Presents an autonomic mechanism for anomaly detection in a compute Cloud system using PCA and Bayesian models for feature extraction and Expectation Maximization clustering algorithm for anomaly detection.
Bhaduri et al. [2011]	K-Nearest Neighbours	Proposes an automated failure detection system employing distance-based anomaly rules to identify faulty machines in a cluster.
Guan and Fu [2013b] and Guan et al. [2013]	Wavelets	Presents a method analyzing performance metrics in both time and frequency domains in order to identify anomalous behaviors in a Cloud environment.
Fu et al. [2012]	Support Vector Machines (SVM)	Proposes an hybrid self-evolving anomaly detection framework using one-class and two-class SVM.

using *Decision Trees*, *Association Rules*, *C4.5* classification. During the testing phase, for each test instance, the best rule that captures the instance is used to compute an anomaly score for designating the test instances as anomalous or normal. A rule has an associated confidence score proportional to the ratio between the number of correct classification by the rule and total number of cases covered by the rule. The anomaly score is computed as the inverse of the confidence score associated with a given rule [Chandola et al. 2009].

The complexity of classification techniques depends on the algorithms used. Training decision trees is often faster than training techniques such as SVM that involves quadratic optimization. The testing phase is also faster. Classification methods rely heavily on accurately labeled data, and also produces class labels which may not be useful in cases where an associated score is required.

3.2.2.2. Neighbour-based Techniques. Unlike classification-based approaches, neighbour-based techniques are unsupervised learning systems that evaluates data instances based on its local neighbourhood. The assumption is that normal data usually occur in dense neighbourhoods while abnormal data occur far from their closest neighbour [Chandola et al. 2009]. It is also required that a distance or similarity measure is estimated between two data instances depending on the data type. Different methods exist for calculating similarity measures such as Euclidean Distance for continuous data and Mutual Information for categorical data. A popular neighbourhood method is the *k*th-Nearest Neighbour which estimates the distance of a given instance to its nearest neighbours and evaluate the distance against a predefined domain specific threshold [Liao and Vemuri 2002; Lazarevic et al. 2003]. The Local Outlier Factor (LOF) algorithm is another neighbour-based technique that detect anomalous instances by estimating the density of each instance. Instances in low-density neighbourhoods are classified as anomalous [Wang et al. 2012]. Basic neighbour-based and LOF methods has a time complexity of $O(N^2)$. Its testing phase is computationally intensive because distance score of a test instance to others is required. It is also difficult to create distance measures for complex data (e.g., spatial and streaming data).

3.2.2.3. Clustering-based Techniques. Clustering is another type of unsupervised learning that groups similar data instances into clusters according to hidden relationships between instances in a cluster [Berkhin 2006]. The goal is to find clusters of similar data points such that each cluster is well separated. Detection of anomalous instances can be based on the density of the clusters (e.g., dense or sparse) or distance of instances from the closest centroid in the cluster [Chandola et al. 2009]. The Euclidean distance, Mahalanobis distance, and Cosine similarity are example of distance measures for such cases. Examples of clustering algorithms include the K-means clustering, Expectation Maximization (EM), and Self-Organizing Maps (SOM) [Hodge and Austin 2004]. Time complexity of clustering depends on the algorithm in use. Testing phase is faster since test instances are compared with only a few cluster.

3.2.2.4. Information Theoretic Techniques. Information theory provides many measures for estimating the degree of dispersal or concentration of the information content of a data set [Wang et al. 2010]. The primary assumption of these methods is that anomalies induce irregularities in the information content of a given dataset [Chandola et al. 2009]. Also they are very generic in nature with no need for parameterization [Wagner and Plattner 2005]. The Entropy information measure or Shannon-Wiener Index [Shannon 2001] estimates the degree of uncertainty in a given dataset. Given a random variable X, its entropy is computed as $H(X) = -\sum_{i=1}^{n} P(x_i)\log(P(x_i))$, where $P(x)$ is the probability distribution of X. The entropy $H(X)$ lies in the range

$[0, \log(n)]$. Higher entropy values indicate more randomness in the data and may be more anomalous than data with lower $H(X)$ values [Navaz et al. 2013]. The degree of randomness between two random variables with probability distributions $P(x)$ and $Q(x)$ can be estimated by their Relative Entropy, $H(Q|P) = \sum Q(x) \log \frac{Q(x)}{P(x)}$. An application of this is to compare the entropy values of two different windows of observation of a metric for detecting changes. The smaller the relative entropy the better. A $H(Q||P)$ value of 0 indicates that the probability distributions $P(X)$ and $Q(X)$ exhibit the same randomness [Lee and Xiang 2001]. Entropy-based methods have been applied to study malicious behaviours in network traffic in Wagner and Plattner [2005] and Lee and Xiang [2001]. Wang et al. [2009, 2010] present entropy-based methodologies for detecting anomalies in a cloud computing environment by analyzing metric distributions. Entropy generally provide more fine-grained insights of the data than traditional classification methods [Nychis et al. 2008] and suitable for online unsupervised detection of unknown anomalies [Wang et al. 2010] since no assumptions of underlying distribution is made.

4. RESEARCH TRENDS

Before the 2000s, contributions focused primarily on the detection of coarse-grained performance issues such as identifying hardware, software bottlenecks in the operating systems [Mahapatra and Venkatrao 1999; Breese and Blake 1995], networks [Melander et al. 2000], and client-server applications [Neilson et al. 1995].

Due to the emergence of the Internet, the early 2000s witnessed a slow trend towards web and distributed applications hosted in dedicated environments. Chen et al. [2002], Aguilera et al. [2003], and Barham et al. [2003] proposed techniques for uncovering performance failures and anomalies with regards to web and distributed systems. By the mid to late 2000s, efforts concentrated on building improved detection mechanisms targeting enterprise applications running in shared-hosting environments, grid, and large-scale infrastructures. This period witnessed the development of analytical approaches and tools such as transaction mix models [Kelly 2005b], queuing-theoretic models [Kelly 2005a], signature models [Mi et al. 2008b; Cherkasova et al. 2008], and statistical techniques [Malkowski et al. 2007; Cherkasova et al. 2009]. While efforts such as in Jung et al. [2006] and Malkowski et al. [2009] propose an experimental approach, Chung et al. [2008] and Agarwala et al. [2007] demonstrate the potential of analyzing the flow of messages across distributed components as a suitable method for detecting performance abnormalities.

From the late 2000s until now, the research contributions have been largely consolidated on achieving dependability [Guan and Fu 2013b; Lee et al. 2012] predictable performance [Tan et al. 2010], root-cause identification [Magalhaes and Moura Silva 2011; Bhaduri et al. 2011; Yu and Lan 2013] and meeting performance guarantees [Kang et al. 2012; Lan et al. 2010] in cloud computing applications and systems. Similarly, there are systems tailored to detecting and resolving workload related anomalies [Wang et al. 2012, 2014]. Perhaps due to scale and the special requirements imposed by the cloud, advanced machine learning techniques have found extensive use in bottleneck and anomaly detection research [Dean et al. 2012, 2014; Huang et al. 2013; Sharma et al. 2013]. Even though existing research contribution is dominated by reactive solutions, there is increasing shift toward proactive approach. Predictive anomaly and bottleneck detection offers better system reliability by raising in advance, just-in-time alerts and detecting potential bottlenecks before a performance issue occur. Examples of such approach can be found in Guan et al. [2011] and Tan and Adviser-Gu [2012].

Following the trends, we observe that cloud computing systems and applications will continue to attract the attention of performance anomaly detection and bottleneck identification research. Characteristics of the cloud systems such as heterogeneity of resources and application services, variable load, and performance variation complicate the problem of detecting performance issue [Gong et al. 2010]. We describe these challenges in detail in Section 5.

5. PADBI SYSTEMS IN THE CLOUD: SPECIFIC REQUIREMENTS

Cloud computing enables computing resources to be provisioned on demand as an utility over the Internet and dynamically scale in response to unpredictable demands and application workloads. A cloud infrastructure is typically characterized by a pool of heterogeneous hardware and software resources that are shared by many application services with disparate performance objectives [Zhang et al. 2010; Jennings and Stadler 2014]. The resulting resource contention and performance interference caused by resource sharing have significant impact on the performance of cloud services and systems [Sharma et al. 2013; Wang et al. 2010].

Inherent characteristics of the cloud such as the heterogeneity of resource types and their interdependencies; the variability and unpredictability of load; and the complex architecture of cloud services; make the task of detecting and resolving performance problems more difficult. To meet stringent performance objectives and to achieve predictable performance, PADBI systems must take into consideration specific cloud requirements as described in the following text.

(1) *Scale.* Medium- to large-scale cloud infrastructures run up to thousands of applications on limited computing resources. It is daunting to keep track of the execution status of such huge applications base [Tan et al. 2012; Dean et al. 2012]. Considering that these applications are composed of multiple service components and the complex topology of the infrastructure, the potential metric space is huge. Wang et al. [2010] estimates this to the Exa scale. That is up to 10^{18} metrics to monitor and process in real time! This require PADBI systems to be lightweight with negligible performance and storage overhead. Also, they must be able to operate in an online fashion in order to keep up with the time varying nature of the cloud.
(2) *Multitenancy.* Multitenancy enables different applications (deployed in virtual machines (VMs)) to be colocated on the same physical server. These VMs concurrently share and compete for virtualized resources (such as CPU and memory) and non virtualized resources (such as network and caches). Such a tight execution environment has been shown to account for 40% in performance degradation in some applications [Sharma et al. 2013]. This makes it essential for PADBI techniques to be aware of prevailing execution contexts.
(3) *Complex Application Architecture.* The cloud run an heterogeneous mix of applications with time-varying workload patterns, ranging from long-running MapReduce jobs and HPC scientific workflows; to interactive web-based social media platforms, e-commerce, and media streaming applications [Wang et al. 2010]. Also, many of these applications share temporal dependency such as two applications having similar workload behaviours. Moreover, services in IaaS clouds come in black-boxes with limited visibility by the cloud infrastructure provider. This limits the extent to which performance degradation issues can be diagnosed and resolved [Dean et al. 2012; Tan et al. 2012].
(4) *Dynamic Resource Management.* Due to the continuous flow of load in and out of the cloud, resource management tasks such as dynamic reconfiguration, consolidation and migration constantly change the operational context in which applications runs

[Tan et al. 2012; Wang et al. 2010]. This leads to a higher frequency of anomalies. Faulty VM reconfigurations, and spontaneous live migrations have been observed to impact performance by up to 30% and 10%, respectively [Sharma et al. 2013]. In such environments, it is nearly difficult to determine what performance behaviour is normal and which is not [Dean et al. 2012].

(5) *Autonomic Management.* Today's data centers are powered by highly automated mechanisms. Autonomic resource managers dynamically provision resources based on adaptive system policies to meet expected quality of service (QoS) and achieve optimal resource utilization levels [Buyya et al. 2012; Hasan et al. 2012]. Delayed detection and manual resolutions do not fit the cloud model, as they can cause prolonged performance violations with huge financial penalty and failure [Dean et al. 2012; Tan et al. 2012]. Therefore, PADBI systems for the cloud are must be dynamic and proactive in nature [Sharma et al. 2013; Wang et al. 2010].

6. DISCUSSIONS AND FUTURE DIRECTIONS

The motivation for detecting unexpected performance behaviours and their root-causes is due to the significant impact they have on smooth operation of systems, the criticality of information they bear, and the costly penalties due to loss of dissatisfied users. The choice of detection is influenced not only by the characteristics of the anomalies and bottlenecks of interest but also by the nature of data and system under test.

PADBI systems based on statistical methods are only as correct as the correctness of the data, the assumption of its distribution and the fitness of the analysis. It is very important to collect the right data and quantity. Care must be taken to balance the proportion of normal samples to anomaly samples in the dataset to avoid the "needle in a haystack" [6] problem. Though parametric techniques assume known data distribution and best at identifying well-known anomalies, nonparametric methods are resistant to high variation in the data without knowledge of data distribution.

Machine learning solutions can quickly sift through a massive metric space to identify patterns of interests or indistinct relationships. Learning techniques expect that normal data instances are more frequent in the data; otherwise, they suffer from high false detection. While most classification, clustering, and statistical techniques have expensive training phases, they provide fast testing with high false-positive detection when unknown anomalous data is frequent. On the other hand, neighbour-based learning methods require no training phase and are highly suitable for real-time detection. However, they are computationally expensive.

Further advancement in hybrid solutions holds great potential for today's system such as proposed in Fu et al. [2012]. Rigid assumptions (regarding distribution and density of performance data) imposed by statistical techniques do not always work in dynamic environments. In addition, unsupervised algorithms are known to perform poorly in cases where anomalies occur more frequently in the test data than normal. When deciding the choice of methods to use in a given case, it is important to consider the tradeoff between online and offline detection as well as the cost incurred when there is a requirement for frequent model updates. Today's systems are dynamic with constant changing execution contexts, application composition, and configurations. It is also expected that anomaly detection and bottleneck identification mechanisms are able to *adapt* as well. Methods that require extensive training phase is inadequate in this case. Focus then must be on techniques that support online updates of model parameters and variables.

[6]A situation where it is nearly impossible to detect anomalous instances in the dataset because only a few anomalous instances exist in the training data.

Tables IX and X of Appendix A summarize major references used in this work based on the essential characteristics of the PADBI problem. Furthermore, we have identified a few promising directions and open challenges within the scope of the problem and briefly outline them in the following text:

(1) *Multilevel bottleneck detection.* Current efforts must extend toward the detection of performance bottlenecks at different levels considering the complexity of today's infrastructure and application. For instance, it should be possible to identify bottlenecks from a set of top-level application service components and further down through the virtualization layer to system resource bottlenecks. Similarly, anomaly detection should be viewed from three perspectives: workload, resource demand and performance.
(2) *Taxonomy of performance bottlenecks and anomalies.* A taxonomy of performance issues under various operational condition (e.g., workload, platform) and manifestation will be highly essential for industry and academia. The challenge here is that these behaviours are inherently intrinsic to the applications and their manifestations vary from one application to another. However, we believe little steps can be made toward this especially for common performance anomalies and bottlenecks. A similar direction is documented in Pertet and Narasimhan [2005].
(3) *Open performance datasets.* Their lack of open performance datasets hinders the pace of research in this area because such data are often considered highly sensitive or classified. Google Cluster [Reiss et al. 2011] trace serves a similar purpose. However, the Google data is an old 29-day trace of a 12,000-machine cluster covering jobs, tasks, resource usage, and machine events measurement from 2011. Similarly, the Yahoo Webscope [Yahoo! 2014] project provides system measurements of the infrastructure running its cloud serving benchmark system [Cooper et al. 2010]. However, the data covers only resource usage across system components over a mere 30-minute period. Due to sensitivity, these datasets do not contain performance metrics such as throughput and latency. Similar lack of dataset for failure detection research is acknowledged by Schroeder et al. [2010].
(4) *Anomaly-resistant resource allocation.* The autonomic nature of modern IT infrastructures demands tight integration of proactive anomaly detection mechanisms with autonomic resource managers. Alerting administrators of an anomaly delays the detection and resolutions of performance problems. This semiautomated approach does not fit today's model of system management, where prolonged performance violations may induce significant unplanned downtimes.
(5) *Context-aware detection.* Frequent performance variations exhibited by cloud applications have been attributed to the changing execution context of the underlying environment. This is often due to frequent workload variation and dynamic resource reconfiguration. The challenge is identifying and characterizing execution contexts as they evolve over time. Context-aware solutions capable of achieving this in addition to adapting to nonstationary cloud behaviours will greatly improve application performance. Tan et al. [2010] and Tan and Gu [2010], and Sharma et al. [2013] present interesting directions in this case.
(6) *Distributed detection.* A huge chunk of current research focus is on centralized detection. Modern enterprise systems are inherently distributed with components spanning multiple physical domains (servers or data centers). Often times the collection of data across such domains is impractical or difficult due to potential system overheads and proprietary and privacy regulations. This implication calls for a decentralized approach that fits naturally with such systems. A theoretical

attempt is presented in Lazarevic et al. [2009], while a similar case study for failure detection is studied in Bhaduri et al. [2011].

7. CONCLUDING REMARKS

We present a review of the performance anomaly detection and bottleneck identification problem and identify relevant research questions, challenges, contributions, trends and open issues. For clarity, we highlight different types of commonly observed performance anomalies and bottlenecks in computing systems. Existing PADBI systems operate based on one or more detection strategies and methods. Statistical and machine learning are the two predominant methods in literature. We have highlighted major classes of techniques in both methods along with interesting references. The choice of strategies and techniques is largely influenced by the goal of the system and the core elements of the problem such as the nature of the system or application, the performance data, and the extent to which the system can be observed. Based on trends, the problem of detecting performance issues and their root-causes will continue to attract research attention, especially in cloud services. We also highlighted specific requirements for effective anomaly and bottleneck detection in cloud computing infrastructures. However, the problem of multilevel bottleneck detection, distributed detection, and accessible performance datasets still remain open research issues.

A. APPENDIX A

B. GENERAL OVERVIEW OF PADBI SYSTEMS

Table IX. Overview of PADBI Systems

Reference	Goal	System	Observability	Strategy	Method	Techniques
Chen et al. [2002]	PBI	Distributed, Web-based, Component bottlenecks	White-box, Source Tracing	Flow & Dependency	Hybrid	Clustering, Correlation
Cohen et al. [2004]	PADBI	Multi-tier, Web-based, System metrics	Black-box, Profiling	Observational	Machine Learning	Tree Augmented Bayesian Networks
Kelly [2005a]	PAD	Distributed, Web-based, Application & System metrics	Gray-box, Profiling	Observational	Statistical	Regression, Transaction mix Model
Cohen et al. [2005]	PAD	Distributed, Enterprise, Application & System metrics	Black-box, Profiling	Signature-based	Machine Learning	Clustering, Tree-Augmented Naive Bayes Models
Jung et al. [2006]	PADBI	Multi-tier, Application & System metrics	Black-box, Profiling	Observational	Machine Learning	C4.5 Decision Tree
Malkowski et al. [2007]	PBI	Web-based, System metrics	Gray-box, Profiling	Knowledge-based	Statistical	SIA

(Continued)

Table IX. Continued

Reference	Goal	System	Observability	Strategy	Method	Techniques
Agarwala et al. [2007]	PADBI	Distributed, Multi-tier, Application & System metrics	White-box	Flow & Dependency	Statistical	Correlation
Zhang et al. [2007b]	PAD	Multi-tier, System & Application metrics	Gray, Profiling	Observational	Statistical	Regression, TM models, Queuing model
Gunter et al. [2007]	PAD	Grid, System metrics	Black-box, Logging	Observational	Statistical	MSD, CDF, EWMA
Yang et al. [2007]	PAD	Grid, System resource metrics	Black-box, Profiling	Observation, Flow & Dependency	Statistical, Signal Processing	Extended Window Averaging, Regression
Chung et al. [2008]	PBI	HPC, Resource bottlenecks	Gray, Profiling	Knowledge-based	-	Inference Engine
Cherkasova [et al. 2008]	PAD	Web-based, Multi-tier, Application & System metrics	Gray, Profiling	Signature-based	Statistical	Regression, Transaction mix Model
Bodík et al. [2008]	PAD	Distributed Systems, System metrics	Black-box, Profiling	Signature-based	Machine Learning	Logistic Regression with L1 Regularization
Malkowski et al. [2009]	PADBI	Multi-tier, Application & System metrics	Gray	Observational	Statistical, Signal Processing	Kernel Density Estimation, Adaptive Filtering
Wang et al. [2009]	PBI	Multi-tier	Gray	Observational	-	Heuristics
Kandula et al. [2009]	PADBI	Enterprise Systems	Gray-box	Knowledge-based, Flow & Dependency	Statistical Learning	Probability Models, Inference Engine
Ben-Yehuda et al. [2009]	PBI	Virtualized, Component & Resource bottlenecks	Black-box, Profiling	Flow & Dependency	Statistical, Queueing Theory	Percentile Testing, Little's Law
Iqbal et al. [2010]	PBI	Cloud, Multi-tier, Resource bottlenecks	Black-box, Profiling	Observational	-	-
Bodik et al. [2010]	PAD	Distributed Systems, Cloud, System metrics	Black-box, Profiling	Signature-based	Statistical and Machine Learning	Quantile Summarization, Logistic Regression with L1 Regularization

Table X. Overview of PADBI Systems (cont.)

Reference	Goal	System	Observability	Strategy	Method	Techniques
Lan et al. [2010]	PAD	Cloud, Host/Node bottlenecks	Black-box, Profiling	Observational	Machine Learning	Principal and Independent Component Analysis
Magalhaes and Moura Silva [2011]	PADBI	Web-based, Application metrics	White-box, Request tracing	Flow & Dependency	Statistical	Correlation Analysis, ANOVA
Fu [2011]	PAD	Cloud, System metrics	Black-box, Profiling	Knowledge-based	Machine Learning	Mutual Information, PCA, Semi-supervised Decision-tree
Sambasivan et al. [2011]	PAD	Distributed, Storage, Network request flows	Gray, Tracing	Flow & Dependency	Hybrid	C4.5, Regression Tree
Tan et al. [2012]	PAD	Cloud, Web-based, System metrics	Black-box, Profiling	Observational	Hybrid	Markov-model, Tree Augmented Bayes
Pannu et al. [2012]	PAD	Cloud, System metrics	Black-box, Profiling	Knowledge-driven	Machine Learning	Supervised, One-class SVM
Lee et al. [2012]	PADBI	Distributed, Storage, Application metrics	Black-box, Profiling	Knowledge-base	Statistical	SPC
Kang et al. [2012]	PADBI	Cloud, Application & System metrics	Black-box, Profiling	Observational	Hybrid	Regression (LAR), Clustering
Dean et al. [2012]	PADBI	Cloud, System metrics	Black-box, Profiling	Observational	Machine Learning	Self-Organizing Maps
Wang et al. [2012]	PAD	Web-based, System metrics	Gray-box, Profiling	Observational	Machine Learning	Clustering, LOF

(Continued)

Table X. Continued

Reference	Goal	System	Observability	Strategy	Method	Techniques
Nguyen et al. [2013]	PADBI	Cloud	Black-box	Flow & Dependency	Statistical, Signal Processing	CUSUM, FFT Filtering
Guan and Fu [2013b]	PAD	Cloud, System metrics	Black-box, Profiling	Observational	Machine Learning	Wavelet, Sliding Window
Wang et al. [2013b]	PBI	Multitier, Component & Resource bottlenecks	Blackbox	Observation	-	Fine-grained Load, Throughput Analysis
Xiong et al. [2013]	PAD	Distributed, Resource metrics	Blackbox, Profiling	Observation	Statistical	Correlation Analysis
Sharma et al. [2013]	PAD	Cloud, Web-based, System metrics	Black-box, Profiling	Observational	Machine Learning	Hidden Markov Model, k-Nearest Neighbour, K-means Clustering
Huang et al. [2013]	PAD	Cloud, System metrics	Black-box, Profiling	Knowledge-based	Machine Learning	LOF
Yu and Lan [2013]	PAD	Distributed, Hadoop Clusters, Component and Host bottlenecks	Black-box, Profiling	Observational	Machine Learning	Non-parametric Clustering
Wang et al. [2014]	PAD	Web-based, System metrics	Gray-box, Profiling	Observation	Machine Learning	Clustering, LOF
Dean et al. [2014]	PADBI	Cloud, System metrics	Black-box, System-call tracing	Observation	Statistical	Tukey Limits

REFERENCES

Sandip Agarwala, Fernando Alegre, Karsten Schwan, and Jegannathan Mehalingham. 2007. E2EProf: Automated end-to-end performance management for enterprise systems. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 749–758.

Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review* 37, 74–89.

E. Alpaydin. 2014. *Introduction to Machine Learning*. MIT Press.

Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*. 85–90.

Roberto Battiti. 1994. Using mutual information for selecting features in supervised neural net learning. *IEEE Transactions on Neural Networks* 5, 4, 537–550.

Muli Ben-Yehuda, David Breitgand, Michael Factor, Hillel Kolodner, Valentin Kravtsov, and Dan Pelleg. 2009. NAP: A building block for remediating performance bottlenecks via black box network analysis. In *Proceedings of the 6th International Conference on Autonomic Computing*. ACM, 179–188.

Frank M. Bereznay and Kaiser Permanente. 2006. Did something change? using statistical techniques to interpret service and resource metrics. In *Proceedings of the International CMG Conference*. 229–242.

Pavel Berkhin. 2006. A survey of clustering data mining techniques. In *Grouping Multidimensional Data*. Springer, 25–71.

Kanishka Bhaduri, Kamalika Das, and Bryan L. Matthews. 2011. Detecting abnormal machine characteristics in cloud infrastructures. In *Proceedings of the IEEE 11th International Conference on Data Mining Workshops (ICDMW'11)*. IEEE, 137–144.

Walter Binder, Jarle Hulaas, and Philippe Moret. 2007. Advanced java bytecode instrumentation. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*. ACM, 135–144.

Peter Bodík, Moises Goldszmidt, and Armando Fox. 2008. HiLighter: Automatically building robust signatures of performance behavior for small-and large-scale systems. In *SysML*. USENIX Association.

Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen. 2010. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems*. ACM, 111–124.

George E. P. Box and George C. Tiao. 1975. Intervention analysis with applications to economic and environmental problems. *J. Amer. Statist. Assoc.* 70, 349, 70–79.

John S. Breese and Russ Blake. 1995. Automating computer bottleneck detection with belief nets. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann, 36–45.

Jack Brey and Rick Sironi. 1990. Managing at the knee of the curve (The use of SPC in managing a data center). In *Proceedings of the International CMG Conference*. 895–901.

Shaun Burke. 2001. Missing values, outliers, robust statistics & non-parametric methods. *LC-GC Europe Online Supplement, Statistics & Data Analysis* 2, 19–24.

Rajkumar Buyya, Rodrigo N. Calheiros, and Xiaorong Li. 2012. Autonomic cloud computing: Open challenges and architectural elements. In *Proceedings of the 3rd International Conference on Emerging Applications of Information Technology (EAIT'12)*. IEEE, 3–10.

Jeffrey P. Buzen and Annie W. Shum. 1995. Masf-multivariate adaptive statistical filtering. In *Proceedings of the International CMG Conference*. 1–10.

Giuliano Casale, Amir Kalbasi, Diwakar Krishnamurthy, and Jerry Rolia. 2009. Automatic stress testing of multi-tier systems by dynamic bottleneck switch generation. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, 20.

Giuliano Casale, Ningfang Mi, Ludmila Cherkasova, and Evgenia Smirni. 2012. Dealing with burstiness in multi-tier applications: Models and their parameterization. *IEEE Transactions on Software Engineering* 38, 5, 1040–1053.

Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)* 41, 3, 15.

Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of International Conference on Dependable Systems and Networks*. IEEE, 595–604.

Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. 2008. Anomaly? application change? or workload change? Towards automated detection of application performance anomaly

and change. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks with FTCS and DCC*. IEEE, 452–461.

Ludmila Cherkasova, Kivanc Ozonat, Ningfang Mi, Julie Symons, and Evgenia Smirni. 2009. Automated anomaly detection and performance modeling of enterprise applications. *ACM Transactions on Computer Systems (TOCS)* 27, 3, 6.

I-Hsin Chung, Guojing Cong, David Klepacki, Simone Sbaraglia, Seetharami Seelam, and Hui-Fang Wen. 2008. A framework for automated performance bottleneck detection. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*. IEEE, 1–7.

Ira Cohen, Jeffrey S. Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. 2004. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI*, Vol. 4. 16–16.

Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. 2005. Capturing, indexing, clustering, and retrieving system history. In *ACM SIGOPS Operating Systems Review*, Vol. 39. ACM, 105–118.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 143–154.

Marc Courtois and Murray Woodside. 2000. Using regression splines for software performance analysis. In *Proceedings of the 2nd International Workshop on Software and Performance*. ACM, 105–114.

Kaustav Das. 2009. Detecting patterns of anomalies. Technical Report CMU-ML-09-101. PhD thesis. Carnegie Mellon University, Department of Machine Learning.

Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. 2012. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th International Conference on Autonomic Computing*. ACM, 191–200.

Daniel J. Dean, Hiep Nguyen, Peipei Wang, and Xiaohui Gu. 2014. 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*. USENIX Association, 16–16.

Anh Vu Do, Junliang Chen, Chen Wang, Young Choon Lee, Albert Y. Zomaya, and Bing Bing Zhou. 2011. Profiling applications for virtual machine placement in clouds. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD'11)*. IEEE, 660–667.

Evolven. 2011. Downtime, Outages and Failures—Understanding Their True Costs. Retrieved March 11, 2015 from http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html.

Imola K. Fodor. 2002. A survey of dimension reduction techniques. Technical Report UCRL-ID-148494. Lawrence Livermore National Laboratory.

Song Fu. 2011. Performance metric selection for autonomic anomaly detection on cloud computing systems. In *Proceedings of the Global Telecommunications Conference (GLOBECOM'11)*. IEEE, 1–5.

Song Fu, Jianguo Liu, and Husanbir Pannu. 2012. A Hybrid anomaly detection framework in cloud computing using one-class and two-class support vector machines. In *Advanced Data Mining and Applications*. Springer, 726–738.

Alessio Gambi and Giovanni Toffetti. 2012. Modeling cloud performance with kriging. In *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 1439–1440.

Chunye Gong, Jie Liu, Qiang Zhang, Haitao Chen, and Zhenghu Gong. 2010. The characteristics of cloud computing. In *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW'10)*. IEEE, 275–279.

Brendan Gregg. 2013. *Systems Performance: Enterprise and the Cloud*. Pearson Education.

Frank E. Grubbs. 1969. Procedures for detecting outlying observations in samples. *Technometrics* 11, 1, 1–21.

Xiaohui Gu and Haixun Wang. 2009. Online anomaly prediction for robust cluster systems. In *Proceedings of the IEEE 25th International Conference on Data Engineering (ICDE'09)*. IEEE, 1000–1011.

Qiang Guan and Song Fu. 2013a. Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures. In *Proceedings of the IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS'13)*. IEEE, 205–214.

Qiang Guan and Song Fu. 2013b. Wavelet-based multi-scale anomaly identification in cloud computing systems. In *Proceedings of the Global Communications Conference (GLOBECOM'13)*. IEEE, 1379–1384.

Qiang Guan, Song Fu, Nathan DeBardeleben, and Sean Blanchard. 2013. Exploring time and frequency domains for accurate and automated anomaly detection in cloud computing systems. In *Proceedings of the IEEE 19th Pacific Rim International Symposium on Dependable Computing (PRDC'13)*. IEEE, 196–205.

Qiang Guan, Ziming Zhang, and Song Fu. 2011. Proactive failure management by integrated unsupervised and semi-supervised learning for dependable cloud systems. In *Proceedings of the 6th International Conference on Availability, Reliability and Security (ARES'11)*. IEEE, 83–90.

Qiang Guan, Ziming Zhang, and Song Fu. 2012. Ensemble of bayesian predictors and decision trees for proactive failure management in cloud computing systems. *Journal of Communications* 7, 1, 52–61.

Dan Gunter, Brian L. Tierney, Aaron Brown, Martin Swany, John Bresnahan, and Jennifer M. Schopf. 2007. Log summarization and anomaly detection for troubleshooting distributed systems. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*. IEEE, 226–234.

Neil J. Gunther. 2004. Benchmarking blunders and things that go bump in the night. *CoRR*. http://arxiv.org/abs/cs.PF/0404043

Neil J. Gunther. 2011. *Analyzing Computer System Performance with Perl:: PDQ*. Springer.

Masum Z. Hasan, Edgar Magana, Alexander Clemm, Lew Tucker, and Sree Lakshmi D. Gudreddi. 2012. Integrated and autonomic cloud resource scaling. In *Proceedings of the Network Operations and Management Symposium (NOMS'12)*. IEEE, 1327–1334.

Victoria J. Hodge and Jim Austin. 2004. A survey of outlier detection methodologies. *Artificial Intelligence Review* 22, 2, 85–126.

Cheng Huang. 2011. Public DNS System and Global Traffic Management. Retrieved April 15, 2014 from http://research.microsoft.com/en-us/um/people/chengh/slides/pubdns11.pptx.pdf.

Su-Yun Huang, Mei-Hsien Lee, and Chuhsing Kate Hsiao. 2006. Kernel canonical correlation analysis and its applications to nonlinear measures of association and test of independence. Institute of Statistical Science: Academia Sinica, Taiwan.

Tian Huang, Yan Zhu, Qiannan Zhang, Yongxin Zhu, Dongyang Wang, Meikang Qiu, and Lei Liu. 2013. An LOF-based adaptive anomaly detection scheme for cloud computing. In *Proceedings of the IEEE 37th Annual Computer Software and Applications Conference Workshops (COMPSACW'13)*. IEEE, 206–211.

Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. 2010. SLA-driven automatic bottleneck detection and resolution for read intensive multi-tier applications hosted on a cloud. In *Advances in Grid and Pervasive Computing*. Springer, 37–46.

Brendan Jennings and Rolf Stadler. 2014. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, 1–53.

Gueyoung Jung, Galen Swint, Jason Parekh, Calton Pu, and Akhil Sahai. 2006. Detecting bottleneck in n-tier it applications through analysis. In *Large Scale Management of Distributed Systems*. Springer, 149–160.

Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Victor Bahl. 2009. Detailed diagnosis in computer networks. In *ACM SIGCOMM*.

Hui Kang, Xiaoyun Zhu, and Jennifer L. Wong. 2012. DAPA: diagnosing application performance anomalies for virtualized infrastructures. In *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. USENIX.

Terence Kelly. 2005a. Detecting performance anomalies in global applications. In *Proceedings of the 2nd Workshop on Real, Large Distributed Systems (WORLDS'05)*.

Terence Kelly. 2005b. Transaction mix performance models: Methods and application to performance anomaly detection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. ACM, 1–3.

Kissmetrics. 2014. How Loading Time Affects Your Bottom Line. Retrieved April 15, 2014 from http://blog.kissmetrics.com/loading-time/.

David Kleinbaum, Lawrence Kupper, Azhar Nizam, and Eli Rosenberg. 2013. *Applied Regression Analysis and Other Multivariable Methods*. Cengage Learning.

Seth Koehler, Greg Stitt, and Alan D. George. 2011. Platform-aware bottleneck detection for reconfigurable computing applications. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 4, 3, 30.

S. B. Kotsiantis. 2007. Supervised Machine Learning: A review of classification techniques. *Informatica* 31, 249–268.

Zhiling Lan, Ziming Zheng, and Yawei Li. 2010. Toward automated anomaly identification in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems* 21, 2, 174–187.

Aleksandar Lazarevic, Levent Ertöz, Vipin Kumar, Aysel Ozgur, and Jaideep Srivastava. 2003. A comparative study of anomaly detection schemes in network intrusion detection. In *Proceedings of SIAM International Conference on Data Mining*. SIAM, 25–36.

Aleksandar Lazarevic, Nisheeth Srivastava, Ashutosh Tiwari, Josh Isom, Nikunj C. Oza, and Jaideep Srivastava. 2009. Theoretically optimal distributed anomaly detection. In *Proceedings of the IEEE International Conference on Data Mining Workshops (ICDMW'09)*. IEEE, 515–520.

Benjamin C. Lee and David M. Brooks. 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 185–194.

Donghun Lee, Sang K. Cha, and Arthur H. Lee. 2012. A performance anomaly detection and analysis framework for DBMS development. *IEEE Transactions on Knowledge and Data Engineering* 24, 8, 1345–1360.

Han Bok Lee and Benjamin G. Zorn. 1997. BIT: A Tool for instrumenting java bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*. 73–82.

Wenke Lee and Dong Xiang. 2001. Information-theoretic measures for anomaly detection. In *Proceedings of IEEE Symposium on Security and Privacy (S&P'01)*. IEEE, 130–143.

Li Li and Allen D. Malony. 2006. Model-based performance diagnosis of master-worker parallel computations. In *Euro-Par 2006 Parallel Processing*. Springer, 35–46.

Yihua Liao and V. Rao Vemuri. 2002. Use of k-nearest neighbor classifier for intrusion detection. *Computers & Security* 21, 5, 439–448.

David J. Lilja. 2005. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press.

Joao Paulo Magalhaes and L. Moura Silva. 2011. Adaptive profiling for root-cause analysis of performance anomalies in web-based applications. In *Proceedings of the 10th IEEE International Symposium on Network Computing and Applications (NCA'11)*. IEEE, 171–178.

Joao Paulo Magalhaes and Luis Moura Silva. 2010. Detection of performance anomalies in web-based applications. In *Proceedings of the 9th IEEE International Symposium on Network Computing and Applications (NCA'10)*. IEEE, 60–67.

João Paulo Magalhães and Luis Moura Silva. 2011. Root-cause analysis of performance anomalies in web-based applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 209–216.

Nihar R. Mahapatra and Balakrishna Venkatrao. 1999. The processor-memory bottleneck: Problems and solutions. *Crossroads* 5, 3es, 2.

Simon Malkowski, Markus Hedwig, Jason Parekh, Calton Pu, and Akhil Sahai. 2007. Bottleneck detection using statistical intervention analysis. In *Managing Virtualization of Networks and Services*. Springer, 122–134.

Simon Malkowski, Markus Hedwig, and Calton Pu. 2009. Experimental evaluation of N-tier systems: Observation and analysis of multi-bottlenecks. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE, 118–127.

Markos Markou and Sameer Singh. 2003. Novelty detection: A review part 1: Statistical approaches. *Signal processing* 83, 12, 2481–2497.

Andrew McHugh. 2013. Top 10 Web Outages of 2013. Retrieved March 11, 2015 from http://blog.smartbear.com/performance/top-10-web-outages-of-2013/.

Bob Melander, Mats Bjorkman, and Per Gunningberg. 2000. A new end-to-end probing and analysis method for estimating bandwidth bottlenecks. In *Proceedings of the Global Telecommunications Conference (GLOBECOM'00). IEEE*, Vol. 1. IEEE, 415–420.

Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. 2008a. Burstiness in multi-tier applications: Symptoms, causes, and new models. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag, New York, 265–286.

Ningfang Mi, Ludmila Cherkasova, Kivanc Ozonat, Julie Symons, and Evgenia Smirni. 2008b. Analysis of application performance and its change via representative application signatures. In *Proceedings of the Network Operations and Management Symposium*. IEEE, 216–223.

Jogesh K. Muppala, Steven P. Woolet, and Kishor S. Trivedi. 1991. Real-time systems performance in the presence of failures. *Computer* 24, 5, 37–47.

A. S. Navaz, V. Sangeetha, and C. Prabhadevi. 2013. Entropy based anomaly detection system to prevent ddos attacks in cloud. *International Journal of Computer Applications (0975-8887)* 62, 15. http://arxiv.org/abs/1308.6745

John E. Neilson, C. Murray Woodside, Dorina C. Petriu, and Shikharesh Majumdar. 1995. Software bottlenecking in client-server systems and rendezvous networks. *IEEE Transactions on Software Engineering* 21, 9, 776–782.

Hiep Nguyen, Zhiming Shen, Yongmin Tan, and Xiaohui Gu. 2013. FChain: Toward black-box online fault localization for cloud systems. In *Proceedings of the IEEE 33rd International Conference on Distributed Computing Systems (ICDCS'13)*. IEEE, 21–30.

George Nychis, Vyas Sekar, David G. Andersen, Hyong Kim, and Hui Zhang. 2008. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM conference on Internet Measurement*. ACM, 151–156.

John S. Oakland. 2008. *Statistical Process control*. Routledge.

Husanbir S. Pannu, Jianguo Liu, and Song Fu. 2012. A self-evolving anomaly detection framework for developing highly dependable utility clouds. In *Proceedings of the Global Communications Conference (GLOBECOM'12)*. IEEE, 1605–1610.

Iakovos Panourgias. 2011. *NUMA Effects on Multicore, Multisocket Systems*. The University of Edinburgh.

Jason Parekh, Gueyoung Jung, Galen Swint, Calton Pu, and Akhil Sahai. 2006. Issues in bottleneck detection in multi-tier enterprise applications. In *Proceedings of the 14th IEEE International Workshop on Quality of Service (IWQoS'06)*. IEEE, 302–303.

Emanuel Parzen. 1962. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 1065–1076.

Johannes Passing. 2005. Profiling, monitoring and tracing in SAP web application server. Seminar Systems Modelling, Hasso Plattner Insitute for Software Systems Engineering.

Manjula Peiris, James H. Hill, Jorgen Thelin, Sergey Bykov, Gabriel Kliot, and Christian Konig. 2014. PAD: Performance anomaly detection in multi-server distributed systems. In *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD'14)*. IEEE.

Soila Pertet and Priya Narasimhan. 2005. Causes of failure in web applications (cmu-pdl-05-109). *Parallel Data Laboratory*, 48.

Rob Powers, Moises Goldszmidt, and Ira Cohen. 2005. Short term performance forecasting in enterprise systems. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. ACM, 801–807.

Calton Pu, Akhil Sahai, Jason Parekh, Gueyoung Jung, Ji Bae, You-Kyung Cha, Timothy Garcia, Danesh Irani, Jae Lee, and Qifeng Lin. 2007. An observation-based approach to performance characterization of distributed n-tier applications. In *IEEE 10th International Symposium on Workload Characterization. IISWC 2007*. IEEE, 161–170.

Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. 2010. Understanding performance interference of i/o workload in virtualized cloud environments. In *Proceedings of the IEEE 3rd International Conference on Cloud Computing (CLOUD'10)*. IEEE, 51–58.

Sutharshan Rajasegarar, Christopher Leckie, and Marimuthu Palaniswami. 2008. Anomaly detection in wireless sensor networks. *Wireless Communications, IEEE* 15, 4, 34–40.

Christoph Rathfelder, Stefan Becker, Klaus Krogmann, and Ralf Reussner. 2012. Workload-aware system monitoring using performance predictions applied to a large-scale e-mail system. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA'12)*. IEEE, 31–40.

Charles Reiss, John Wilkes, and Joseph L. Hellerstein. 2011. Google cluster-usage traces: Format+ schema. *Google Inc., White Paper*.

Douglas Reynolds. 2009. Gaussian mixture models. *Encyclopedia of Biometrics*, 659–663.

S. Rogers and M. Girolami. 2011. *A First Course in Machine Learning*. Taylor & Francis.

Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 43–56.

Bianca Schroeder, Garth Gibson, and others. 2010. A Large-scale study of failures in high-performance-computing systems. *IEEE Transactions on Dependable and Secure Computing* 7, 4, 337–350.

Craig A. Shallahamer. 1995. Predicting Computing System Capacity and Throughput. Oracle Corporation White Paper. Retrieved from http://www.orapub.com.

Claude Elwood Shannon. 2001. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review* 5, 1, 3–55.

Bikash Sharma, Praveen Jayachandran, Akshat Verma, and Chita R. Das. 2013. CloudPD: Problem determination and diagnosis in shared dynamic clouds. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*. IEEE, 1–12.

Sameer Shende. 1999. Profiling and tracing in Linux. In *Proceedings of the Extreme Linux Workshop*, Vol. 2. Citeseer.

Derek Smith, Qiang Guan, and Song Fu. 2010. An anomaly detection framework for autonomic management of compute cloud systems. In *Proceedings of the IEEE 34th Annual Computer Software and Applications Conference Workshops (COMPSACW'10)*. IEEE, 376–381.

Ralf Steuer, Jürgen Kurths, Carsten O. Daub, Janko Weise, and Joachim Selbig. 2002. The mutual information: Detecting and evaluating dependencies between variables. *Bioinformatics* 18, Suppl 2, S231–S240.

Yongmin Tan and Xiaohui Helen Adviser-Gu. 2012. *Online Performance Anomaly Prediction and Prevention for Complex Distributed Systems*. North Carolina State University.

Yongmin Tan and Xiaohui Gu. 2010. On predictability of system anomalies in real world. In *Proceedings of the IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'10)*. IEEE, 133–140.

Yongmin Tan, Xiaohui Gu, and Haixun Wang. 2010. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. ACM, 173–182.

Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. 2012. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Proceedings of the IEEE 32nd International Conference on Distributed Computing Systems (ICDCS'12)*. IEEE, 285–294.

Jean-Claude Tarby, Houcine Ezzedine, José Rouillard, Chi Dung Tran, Philippe Laporte, and Christophe Kolski. 2007. Traces using aspect oriented programming and interactive agent-based architecture for early usability evaluation: Basic principles and comparison. In *Human-Computer Interaction. Interaction Design and Usability*. Springer, 632–641.

Igor Trubin. 2005. Capturing workload pathology by statistical exception detection system. In *Proceedings of the Computer Measurement Group*. Citeseer.

Igor A. Trubin and Linwood Merritt. 2004. Mainframe global and workload level statistical exception detection system, based on MASF. In *Proceedings of the International CMG Conference*. 671–678.

John Wilder. 1977. Exploratory data analysis. Addison-Wesley, Reading, Mass.

Arno Wagner and Bernhard Plattner. 2005. Entropy based worm and anomaly detection in fast IP networks. In *14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*. IEEE, 172–177.

Christian Walck. 2007. Handbook on statistical distributions for experimentalists. Internal Report SUF-PFY/96-01, University of Stockholm.

Chengwei Wang, Karsten Schwan, and Matthew Wolf. 2009. Ebat: An entropy based online anomaly tester for data center management. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management-Workshops*. IEEE, 79–80.

Chengwei Wang, Vanish Talwar, Karsten Schwan, and Parthasarathy Ranganathan. 2010. Online detection of utility cloud anomalies using metric distributions. In *Proceedings of the Network Operations and Management Symposium (NOMS'10)*. IEEE, 96–103.

Chengwei Wang, Krishnamurthy Viswanathan, Lakshminarayan Choudur, Vanish Talwar, Wade Satterfield, and Karsten Schwan. 2011. Statistical techniques for online anomaly detection in data centers. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM'11)*. IEEE, 385–392.

Haichuan Wang, Qiming Teng, Xiao Zhong, and Peter F. Sweeney. 2009. Understanding cross-tier delay of multi-tier application using selective invocation context extraction. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag, New York, 34.

Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. 2013a. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *Proceedings of the IEEE 33rd International Conference on Distributed Computing Systems (ICDCS'13)*. IEEE, 31–40.

Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. 2013b. An experimental study of rapidly alternating bottlenecks in n-tier applications. In *Proceedings of the IEEE 6th International Conference on Cloud Computing (CLOUD'13)*. IEEE, 171–178.

Tao Wang, Jun Wei, Feng Qin, WenBo Zhang, Hua Zhong, and Tao Huang. 2013. Detecting performance anomaly with correlation analysis for Internetware. *Science China Information Sciences* 56, 8, 1–15.

Tao Wang, Jun Wei, Wenbo Zhang, Hua Zhong, and Tao Huang. 2014. Workload-aware anomaly detection for web applications. *Journal of Systems and Software* 89, 19–32.

Tao Wang, Wenbo Zhang, Jun Wei, and Hua Zhong. 2012. Workload-aware online anomaly detection in enterprise applications with local outlier factor. In *Proceedings of the IEEE 36th Annual Computer Software and Applications Conference (COMPSAC'12)*. IEEE, 25–34.

Pengcheng Xiong, Calton Pu, Xiaoyun Zhu, and Rean Griffith. 2013. vPerfGuard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ACM, 271–282.

Yahoo! 2014. Webscope dataset—Computer System Data. Retrieved from http://webscope.sandbox.yahoo.com/catalog.php?datatype=s.

Lingyun Yang, Chuang Liu, Jennifer M. Schopf, and Ian Foster. 2007. Anomaly detection and diagnosis in grid environments. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 1–9.

Li Yu and Zhiling Lan. 2013. A scalable, non-parametric anomaly detection framework for Hadoop. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 22.

Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. 2011. Profiling network performance for multi-tier data center applications. In *Proceedings of Symposium on Networked System Design and Implementation*. 57–70.

Qi Zhang, Lu Cheng, and Raouf Boutaba. 2010. Cloud computing: State-of-the-art and research challenges. *Journal of Internet Services and Applications* 1, 1, 7–18.

Qi Zhang, Ludmila Cherkasova, Guy Mathews, Wayne Greene, and Evgenia Smirni. 2007b. R-capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads. In *Middleware 2007*. Springer, 244–265.

Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. 2007a. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the 4th International Conference on Autonomic Computing (ICAC'07)*. IEEE, 27–27.

Steve Zhang, Ira Cohen, Moises Goldszmidt, Julie Symons, and Armando Fox. 2005. Ensembles of models for automated diagnosis of system performance problems. In *Proceedings of International Conference on Dependable Systems and Networks*. IEEE, 644–653.

Received December 2014; revised March 2015; accepted May 2015

II

Paper II

Apex Lake: A Framework for Enabling Smart Orchestration

Thijs Metsch[*], Olumuyiwa Ibidunmoye[**],
Victor Bayon-Molino[*], Joe Butler[*],
Francisco Hernandez-Rodriguez[**], Erik Elmroth[**]

[*]*Intel Labs Europe*
Collinstown Industrial Park, Leixlip, Ireland
{*thijs.metsch, victor.bayon-molino, joe.m.butler*}*@intel.com*

[**]*Dept. Computing Science, Umeå University*
SE-901 87 Umeå, Sweden
{*muyi, francisco, elmroth*}*@cs.umu.se*
http://www.cs.umu.se/ds

Abstract: The introduction of a Software-defined infrastructures brings additional challenges to the management of cloud infrastructure. With the impending convergence of telecommunications and cloud infrastructures, datacenters become an essential part of an overall integrated environment. The potential scale of such environments has significant implications as traditional orchestration approaches cannot scale appropriately. However, the combination of infrastructure topology, fine-grained operational data and advanced analytics, has the potential to deliver a scalable approach to facilitate orchestration and resource management. In this paper we introduce Apex Lake, a framework designed to address the question of "how to efficiently define and maintain a physical and logical resource and service landscape enriched by operational data, to support orchestration for optimized service delivery?" We also demonstrate with a use-case illustrating how functionalities provided by Apex Lake can be used dealing with performance anomalies.

Apex Lake: A Framework for Enabling Smart Orchestration

Thijs Metsch
Intel Labs Europe
Collinstown Industrial Park
Leixlip, Ireland
thijs.metsch@intel.com

Olumuyiwa Ibidunmoye
Dept. of Computing Science
Umeå University
Umeå 90187, Sweden
muyi@cs.umu.se

Victor Bayon-Molino
Intel Labs Europe
Collinstown Industrial Park
Leixlip, Ireland
victor.bayon-molino@intel.com

Joe Butler
Intel Labs Europe
Collinstown Industrial Park
Leixlip, Ireland
joe.m.butler@intel.com

Francisco Hernández-Rodriguez
Dept. of Computing Science
Umeå University
Umeå 90187, Sweden
francisco@cs.umu.se

Erik Elmroth
Dept. of Computing Science
Umeå University
Umeå 90187, Sweden
elmroth@cs.umu.se

ABSTRACT

The introduction of a Software-defined infrastructures brings additional challenges to the management of cloud infrastructure. With the impending convergence of telecommunications and cloud infrastructures, datacenters become an essential part of an overall integrated environment. The potential scale of such environments has significant implications as traditional orchestration approaches cannot scale appropriately. However, the combination of infrastructure topology, fine-grained operational data and advanced analytics, has the potential to deliver a scalable approach to facilitate orchestration and resource management. In this paper we introduce Apex Lake, a framework designed to address the question of "how to efficiently define and maintain a physical and logical resource and service landscape enriched by operational data, to support orchestration for optimized service delivery?" We also demonstrate with a use-case illustrating how functionalities provided by Apex Lake can be used dealing with performance anomalies.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer Communication Networks-Distributed Systems; K.6.4 [**Computing Milieux**]: Management of Computing and Information Systems, System Management

General Terms

Management, Measurement, Performance

Keywords

Cloud monitoring and orchestration, Resource Management, Datacenter Management, Software-defined Infrastructure

Middleware Industry '15, December 07-11 2015, Vancouver, BC, Canada.

ACM 978-1-4503-3727-4/15/12...$15.00
http://dx.doi.org/10.1145/2830013.2830016.

1. INTRODUCTION

Increase in cloud adoptions, the proliferation of mobile communication and impending convergence of telecommunications and cloud infrastructures is driving massive growth in the size and capability of today's IT infrastructures [1]. This trend is further fueled by the reduction in capital expenditure and increasing capacity. Thus the task of managing datacenter resources and large-scale applications is becoming increasingly complex due to scale, demands on robustness, performance, and efficiency.

The leading edge of today's cloud resource management embrace two distinctly different approaches. On one hand, are those focusing on resource management, resulting in sophisticated solutions to highly granular subproblems such as, capacity scaling [2], VM scheduling [3], server consolidation [4], etc, sometimes taking service component dependencies into account [5]. The resulting systems are typically based on very specific but high-level monitoring information, such as CPU load or service request rate. On the other hand, there are systems that are able to provide robust resource management at large scales, but in order to achieve the necessary scale they, adhere to more simplistic management methods [6, 7]. Design objectives are primarily focused on providing robust functionality and systems whose behavior is easy to understand and configure rather than to base individual decisions on deep analysis and predictions.

To combine the strengths of both perspectives, there is a need to aggregate low- and high-level monitoring data, with topology information, extract appropriate insights (such as identifying correlations– in space and time, and causalities in performance events,) and to make this knowledge available to resource managers in time. This is at the core of what Apex Lake does by combining topology information with multi-hertz monitoring data and providing versioned views on that to support orchestration in large scale environments such as Software-defined infrastructures (SDI).

1.1 Problem Statement – Orchestration in Software-defined infrastructures

In current cloud infrastructures, virtualization logically

divides physical hardware into smaller, abstracted, containers allowing different means to control the system. Similarly, Software-defined Network (SDN) decouples the data plane from the control plane while Software-defined Storage (SDS) permits managing storage systems independently from the hardware [8]. Each approach provides a control plane to a specific subsystem, thus leading to increase in the number of the control functions (or actuation points). As a result, the question emerges with regards to what control functions are suitable and what interdependencies exist between them? Fortunately these sub-systems have one thing in common which is that they are amenable to instrumentation-oriented monitoring. Therefore their behavior - albeit on a high level, can be understood through the application of appropriate analytics. However as each sub-system is not used in isolation it is crucial to understand their composition and the entities they control.

Orchestration is an approach to automatically[1] and at scale coordinate and control complex management tasks [9]. We use the term *Service Orchestration* to describe the orchestration of services in an infrastructure while *Infrastructure Orchestration* as the orchestration of the infrastructure itself. The increasing scale of systems makes effective orchestration more challenging in large heterogeneous systems. Especially for service orchestration, components need to support control from edge devices (smartphones, IoT sensors etc.) to services hosted in (hybrid) clouds. The implication is that the resources become more heterogeneous in nature. For example, a simple architecture may consists mobile devices at the edge connected to a compute platform running big data and HPC applications on heterogeneous hardware.

In order to address these issues, the Apex Lake framework is developed to support orchestration (and related management tasks) by providing extensive monitoring data; structuring the information about hardware, virtual, and service entities in a dynamic information repository; and providing insights at decision points through an analytics engine. In Section 2, we present an overview of the system.

2. APEX LAKE

Apex Lake is a research framework developed at the Cloud Services Lab, Intel Labs Europe, with a clear vision of efficiently defining and maintaining a landscape of the physical and logical resources and associated services enriched by end-to-end monitoring, to support setup and run-time orchestration for optimized service delivery as shown in Figure 1.

The key goal is to provide a framework which enables numerous orchestration oriented use-cases in SDIs. In general such use-cases can be categorized to those dealing with initial placement decisions (e.g. placement of complex services), re-balancing actuations (e.g. migrating service instance to ensure SLA compliance), and capacity planning (e.g. forklift upgrades of the hardware). Our empirical analysis identified a commonality in required functionalities to achieve this, namely: a full-stack telemetry system, an information repository and an analytics pipeline which collectively form the core functional components of Apex Lake. The framework is loosely coupled to existing cloud orchestration and resource management frameworks such as OpenStack[2], Apache Mesos[3] or Kubernetes[4].

[1]Automation does not take the administrator out of the loop - results of optimizations can be presented to humans for active guidance.

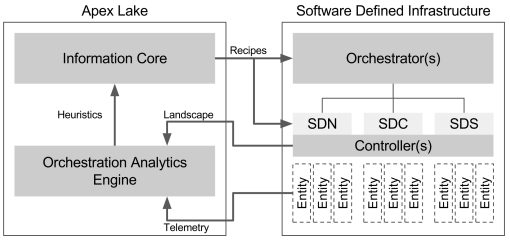


Figure 1: Apex Lake overview.

The following subsections describe the three functional components of Apex Lake.

2.1 Telemetry: Full stack instrumentation

Today, system administrators use a combination of diverse monitoring tools to extract data from the different layers within the operating system. The number of software layers across the stack has also increased by a factor of $\sim 2^3$ in virtualized servers [10]. Moreover, data from the different layers need to be synchronized, integrated, preprocessed, analyzed and presented to users or machine. Therefore full-stack visibility and cross-layer analysis is crucial.

To meet these requirements a robust telemetry framework called *Cimmaron* has been implemented. Cimmaron is a scalable and integrated instrumentation framework allowing users to deploy highly configurable and advanced instrumentation agents coupled with analysis and visualization functionality tailored for performance evaluation workflows. The core functionalities are in two folds. Instrumentation and monitoring– extensive monitoring of both kernel and user spaces, encompassing application and service metrics. It integrates time-series metric data from multiple sources, filters, aggregates and performs data preprocessing at source, in-transit, or at the point of final data aggregation.

Cimmaron's high-resolution and low-overhead event data collection is vital to the orchestration system. It is used to collect extensive data ranging from service-level metrics (per-query latency), system metrics (e.g. per-core, per-socket utilization), hardware counters (e.g. Intel PCM) and environmental metrics (e.g. temperature and energy consumption). In Figure 2, a typical deployment architecture of Cimmaron with scalable back-ends, a control interface, and a visualization front-end is shown.

Cimmaron also offers some key features that makes it suitable for large-scale instrumentation and performance analysis. It provides a unified data storage layer that can be navigated or consumed uniformly. Existing specialized monitoring tools can be connected as controllable plugins. Additionallly, operational parameters and configurations can be modified at runtime. An extensible (and pluggable) set of statistical analysis and graphical charts are available via a web-based user interface to view in real-time system behaviors, discover relationships, detect performance issues

[2]`http://www.openstack.org`
[3]`https://mesos.apache.org/`
[4]`http://www.kubernetes.io`

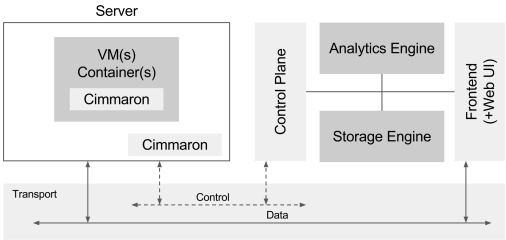


Figure 2: Cimmaron's deployment architecture.

and/or bottlenecks.

2.2 The Information Core

The information core (*Info Core*) is a dynamic repository encapsulating the datacenter as a set of multidimensional planes. The primary responsibility of the Info Core is, to manage the configuration, attributes and spatial information of datacenter entities. This enables us to link the physical and logical topology of the infrastructure with contextual information such as availability, utilization patterns, etc. The Info Core is made up of two parts namely; the landscape; and a contextual database of the landscape. We present them in sections 2.2.1 and 2.2.2.

2.2.1 *Landscape*

The *landscape* is a graph-based model representing the topology and composition of the SDI, its resources and hosted services. Live data from OpenStack is used to organize the entities into three major layers as shown in Figure 3. All hosted application services and their components currently running on the infrastructure are aggregated in the *Service Layer*. The *Virtual Layer* maps all active software-defined entities (e.g. virtual machines, containers, networks ports and block storage). The *Physical Layer* contains all physical resources (e.g. RAM, disks, CPUs, NICs switches, etc).

The landscape is automatically and continuously updated, versioned and maintained in a concurrent state with the SDI. Subgraphs can be extracted from the overall landscape to capture specific scopes within the SDI at any point in time. For example, a service subgraph can be used to identify the components of the service within each layer. Given a service subgraph, it is possible to characterize service performance and identify issues such as interference with other services sharing the same set of resources at the virtual and physical layers.

Nodes of a landscape graph are identified by a set of immutable properties (such as *name*, *type*, *layer* and, *category*) together with mutable state information (such as VM capacity, MAC and IP address, CPU flags, etc.). Edges represent semantic relationships or associations between nodes across or within layers. We currently implement five edge labels such as *depends_on*, *requires*, and *internal* for intra-layer relationships, and *runs_on* and *deployed_on* for inter-layer relationships.

In general, the landscape is useful for understanding the topology and dynamic composition of the SDI, and for tracking its evolution over time. It captures the key features of each entity, its KPIs as well as capacities. More importantly, such data can easily be interpreted to support informed orchestration and to identify performance problems (e.g. to identify performance interference). When enriched with telemetry data the landscape offers multiple planes on which we can identify hot- and/or cold-spots in the infrastructure.

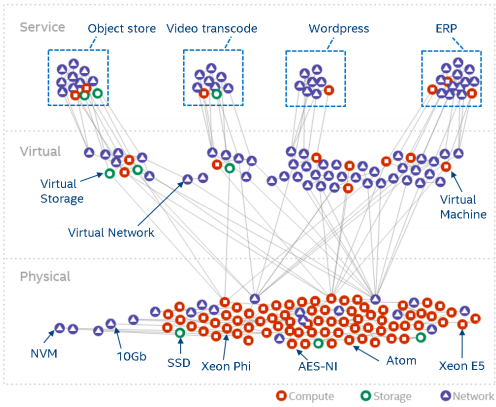


Figure 3: Layers within an E2E landscape.

2.2.2 *Contextual Information*

The purpose of contextual information is to capture the temporal nature of telemetry data and landscape models. Apex Lake provides three types of such information.

Fingerprints are compact, versioned, representations of the runtime behaviours of entities in a SDI. A fingerprint is a tuple $f = (g, Q, t)$ where g is a service subgraph, Q is the set of entity states of all nodes in g, and t is a time-stamp. At every sampling interval, a snapshot of a subgraph is captured and for each node, its state q is appended to Q. An entity state is made up of measured Utilization, Saturation, and Error (see the USE methodology in [11]) rates derived through summary statistics (e.g. average, median or histogram). A time-series of fingerprints are thus suitable for characterizing the behaviour of the SDI at specific points in time or to reason over its evolution.

Heuristics and models are derived by applying machine learning and data mining algorithms to telemetry and landscape data. Models can be used to express, for example, degradation of disks or performance impact of an Intel® Xeon® E5 CPU versus E7. Models are expressed using indicators: $I = F(X|Y)$ which maps to indicator I a function of parameter X given Y. The parameter Y holds information about the landscape and the entities within.

Recipes are used to translate insights (in form of heuristics and models) back into the orchestrator. They are composed of simple *process flows* statements that enable orchestrators and controllers to automatically make decisions. Because they are versioned, different recipes are enabled for different decision points and time. For example, outcomes of scheduling decisions may vary according to the time of the day.

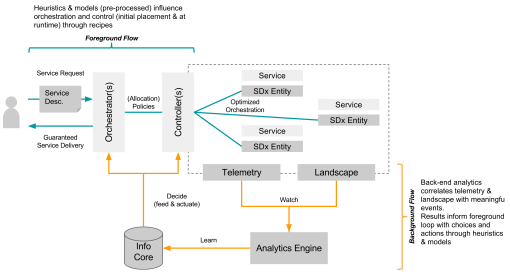


Figure 4: Foreground/Background flows in an Apex Lake enabled environment

2.3 Analytics Engine

The analytics engine is implemented as a cloud-native PaaS service providing a plugable framework for analyzing telemetry and landscape data to produce heuristics and models. It interfaces with Cimmaron and the Info Core via a set of RESTful APIs. Analytics procedures can be implemented as *workbooks* using the Python[5] programming language however they can also be easily integrated with statistical computing environments such as R[6]. Such workbooks are used to analyze incoming data in a manner that is similar to complex event processing systems to generate heuristics and models. Recipes are derived from these heuristics and models to drive decisions that can trigger actuation and control functions within the underlying orchestration and resource management framework. This is achieved through the (internal and remote) APIs of the orchestrators and controllers or with the aid of standards such as OCCI[12].

2.3.1 Analytics for Orchestration

In SDIs, a major concern relates to the feasibility of instrumentation considering the ever increasing depth across infrastructure and services. Apex Lake's solution to this question is based on Netflix's [13] approach. This approach supports inspection of the system through multiple levels of magnifications in order to access different levels of detail. Consequently Cimmaron, the landscape, and Info Core provide views with configurable granularity. The **10x** magnification provides high-level views such as service subgraphs. At **100x**, it is possible to zoom into any node in a subgraph to access finer detail about performance and operational state of the entities. The **1000x** provides the finest view of the landscape. At this level it is possible to drill-down into individual entities to evaluate their behaviors.

Data and control signals flow through Apex Lake via two basic process flows; the *foreground* and *background* flows.

Foreground Flow includes a set of steps that are used to receive and process front-facing events such as processing a customer's service requests or change requests as shown in Figure 4. It is also designated as the *fast loop*.

Background Flow supports deriving contextual information such as the retraining of heuristics and models. As processing of the data can take significant amount of time, this flow is designated the *slow loop*. The slow loop follows a Watch-Learn-Decide cycle as in Figure 4.

It is important to determine which decisions can be enhanced using the process flows. The life-cycle of services (and sub-components) offers a good starting point. A service life-cycle from initial request through provisioning to retirement has numerous decision points associated with it. Since orchestration spans all these phases, it is crucial to determine which decisions can be enhanced by results of analytics. This can be a) for service orchestration such as scaling as well as b) for infrastructure orchestration such as re-balancing the landscape.

In general, there exists many use cases for analytics-based orchestration as provided by Apex Lake. A use case is presented in Section 3, showing how functionalities such as telemetry, information core and analytics are combined to achieve intuitive just-in-time problem identification.

2.4 Deployment

The landscape is stored in a Neo4j[7] graph database instance. To demonstrate the scale of the landscape: a simple OpenStack testbed with 50 active VMs results in a landscape with approximately 1000 nodes describing compute, network and storage resources, within the three layers. As changes in the landscape are tracked, versioning information over a one week time-frame scales the graph to excess of 20,000 nodes. For the same setup, Cimmaron generates over 12,000 data points per second. The data is stored in a HBase[8] cluster with query support via an OpenTSDB[9] datastore. Because Neo4j currently provides no feature to store versioned graphs, contextualized fingerprints are stored in a NoSQL database provided by RethinkDB[10]. Empirically, Apex Lake has low runtime performance overhead (compute and memory) on the underlying system. Cimmaron agent for example only accounts for less than 1 percent of CPU utilization.

3. USE CASE: LANDSCAPE COLOURING

Systems administrators typically navigate and analyze huge volume of data to diagnose performance issue such as resource bottlenecks and failures[14]. Typically, this type of data does not include topology information. Landscape colouring leverages the functionalities provided by Apex lake to classify landscape nodes based on their prevailing states. Machine readable coloured graphs or their subgraphs are then used for detecting anomalous changes in the states of entities. Due to the sheer size of landscape graphs, multiple nodes may be classified anomalous concurrently. In such cases anomalous nodes are ranked to identify high-risk ones.

This use case was conducted on an OpenStack testbed comprising four application services deployed as virtual machines on two physical servers with different configurations. First, using the entity definitions within OpenStack, a full-stack graph G of the landscape is constructed. G is an incomplete digraph having 21, 22, and 38 nodes in the service, virtual and physical layers respectively. The total number of edges is 102 and with a density of 0.02. For this experiment

[5]Python offers a wide varieties of packages for analyzing different kinds of data. Examples are *numpy*, *python pandas*, *scikit-learn*, *r2py*, *scipy*, *networkx*, *statsmodels*, etc

[6]`http://www.r-project.org/`

[7]`http://neo4j.com/`

[8]`http://hbase.apache.org/`

[9]`http://opentsdb.net/`

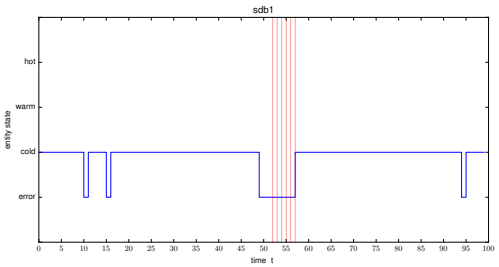
[10]`http://rethinkdb.com/`

we injected three classical performance issues namely disk failure, CPU saturation and network saturation.

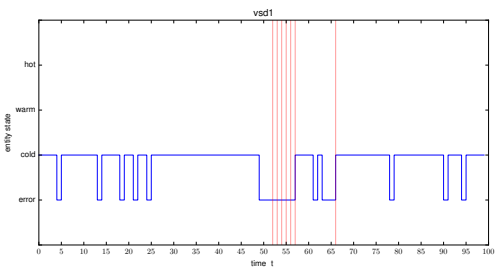
Characterizing entity behaviours. The state of an entity (e.g. compute, network, memory, and storage) in a landscape at time t is a tuple of its utilization and saturation, $y_t = (u_t, s_t)$ derived from summary statistics. Consequently, a rule-based mechanism assigns y_t to a state according to which regions it falls in a two-dimensional map of s_t against u_t. The *error* state may correspond to a faulty entity such as a disk failure characterized by a high queuing rates and an unusually low utilization. The *hot* state is indicative of an over-utilized entity characterized by high queuing and utilization values. An under-utilized landscape entity with low queuing and utilization rates falls into the *cold* region while the *warm* state is characterized by moderate to high utilization and low saturation rates. The bounds of states are parameterized by empirically threshold.

Detecting anomalous entities. The severity of an entity remaining in an error or hot state for long is greater than the cold and warm states. Hence, the error and hot states are classified as *undesirable* while the cold and warm states are classified as *desirable*. A threshold-based change detection algorithm is applied to identify anomalous entities. The threshold, a *hanging length*, l, is the maximum amount of time an entity is allowed to remain in the undesirable state. Overtime, the state of each entity is monitored against the hanging length. Alarms are raised when the threshold is broken. To improve speed and robustness, a careful choice of l is desired so that l is large enough to prevent false alarms and small enough to accommodate transient events.
Figure 5 shows alarm points for a landscape whose zoomed-in snapshot is shown in Figure 6(a). The plots demonstrate the detection of a cascading disk failure in a virtual layer storage entity, vsd_1 deployed on sdb_1, a physical storage device. With hanging length $l = 3$, sdb_1 is first flagged at $t = 53$ and remains so until $t = 57$ when it is restored. Corresponding alarms are observed for vsd_1 and other virtual storage entities due to dependency on sdb_1. Once entities are flagged as anomalous, they are reflected in bold colors in the landscape graph (see Figure 6(b)).

Ranking anomalous entities. Due to the size of the landscape, multiple alarms are possible which can easily overwhelm administrators and management tools (Figure 6(b)). Moreover, the criticality of alarms, should they be rectified later, differ depending on the influence of the entities. Influential entities in a landscape graph are those high-risk nodes with high upper-layer dependencies. Such nodes are similar to hub and authority nodes in web information retrieval. Hence it is necessary to prioritize anomalous entities according to their influence. Our approach exploits the inherent structure of landscape graph itself to identify correlations and dependencies within and across layers. The ranking of anomalous entities is based on the classic *PageRank*[15] algorithm. Given a colored landscape digraph, $G' = (V, E)$, the influence rank of an entity v is computed as $ir(v) = (1-\alpha) + \alpha \sum_{v' \in V'} \frac{ir(v')}{outdeg(v')}$. $v' \in V'$ containing all entities that depends on v. With $outdeg(v')$, each entity spreads its influence out evenly among all entities they depend on. The share of influence contributed by v' to v is given by $\sum_{v' \in V'} \frac{ir(v')}{outdeg(v')}$. The individual influence contributed by each entity depending on v is "damped down" by the parameter $0 < \alpha < 1$, set to 0.85 in this case.



(a) Physical layer storage entity: sdb_1



(b) Virtual layer storage entity: vsd_1

Figure 5: Detection of cascading anomalies

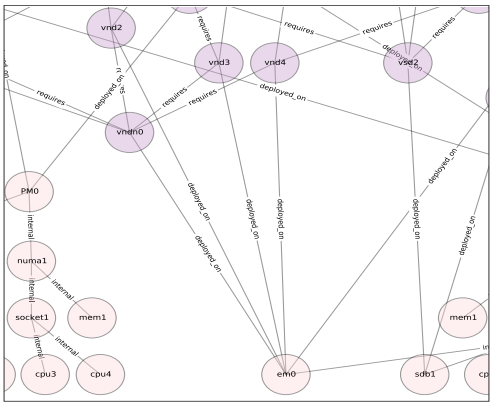
The $(1-\alpha)$ term makes up for entities in the leaves of the physical layer having no out-going links. Ranking results are automatically generated as in Figure 7.

Finally, the influence ranks are then converted to recipes that instruct the orchestrator on how to resolve the anomalies[11]. For example, Figure 7 shows that the network device em_0 is the most influential at the given time followed by sdb_1 and $vndn_0$ respectively. The colors and sizes of points on the graph correspond to their influence.

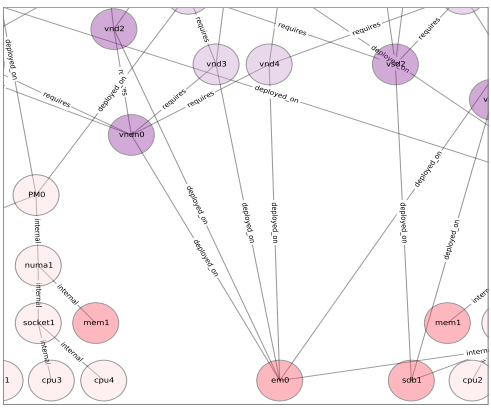
Enhancing orchestration with Landscape Colouring. Landscape colouring is a completely automatic process that can be used as a starting point for root-cause analysis of performance and fault issues in the datacenter. Apex Lake is configured to provide colouring continuously or selectively either for the whole landscape or its subset in the background loop of Figure 4. By continuously monitoring and coloring the landscape a view is created which represents the prevailing states of entities in the landscape and their dependencies. This view shows the implications of nodes on their neighbors. A node and its neighbors might be managed by different subsystems (like SDN or SDS) but thanks to Apex Lake their interplay can be shown and accounted for.

By creating this view and storing it in the Info Core as contextual information, orchestration components are enriched with appropriate just-in-time information. This view can be utilized as an input parameter at multiple actuation points. In order to rectify observed anomalies, the orches-

[11]Coloured landscapes can also serve a basis for visual inspection in smaller landscapes such as scoped or service subgraphs



(a) Initial landscape $t = 0$



(b) Colored landscape at $t = 53$

Figure 6: Testbed landscapes

trator may take scaling or re-balancing decisions and trigger actuation actions to realized required changes. The orchestrator can also use this information to steer new workloads away from physical layer entities exhibiting intermittent bottlenecks behaviors.

4. RELATED WORK

Components of the Apex Lake framework, such as the analytics pipeline, are designed to optimize orchestration in OpenStack and Mesos as well as for realizing demand predictions such as AGILE [16] and SCADS [17].

Nagios and Ganglia already provide monitoring in distributed environments. Closely related to Cimmaron's scalable monitoring and data management is Netflix's Atlas [18] albeit with emphasis on visualization and query optimizations. Our graph-based landscapes extends beyond service and network layer components as in [19] to the architectural level. Although the concept of fingerprints have been used in diverse systems domains for characterizing operational

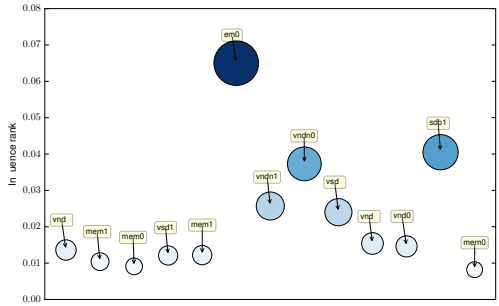


Figure 7: Ranking simultaneous anomalous entities

states [20], fingerprints in Apex Lake extends this idea with topology information to easily assess time-varying dependencies.

5. CONCLUSIONS

The realization of smart orchestration to support Software Defined Infrastructures is predicated on the availability of an integrated platform offering 1) a scalable data collection system, 2) a robust information repository and 3) a seamless analytic service. In this paper, we have demonstrated the viability of Apex Lake to realize this goal. A general overview of the framework and its component parts (i.e. Telemetry, Information Core, and Analytics) has been presented. In particular, the use of rich multi-hertz telemetry data to support contextualization of infrastructure and service landscapes from which heuristics and models are learned is described. Finally, a use-case demonstrating how Apex Lake supports intelligent root-cause analysis for detecting and ranking anomalous SDI entities is outlined.

Apex Lake is being developed by Intel Labs Europe's Cloud Services Lab, Ireland, since 2014. Umeå University made use of Apex Lake to demonstrate the concept of landscape colouring.

A number of use cases for future work have also been defined, which will drive the evolution of the Apex Lake framework. Examples include a) use cases which extend beyond the datacenter to the edge of SDIs such as IoT scenarios b) analysis of interdependent policy enforcement on subsystems which have dependencies on each other (such as SDN and SDS) c) development of control-theoretic autonomics for meeting system-wide objectives.

6. ACKNOWLEDGMENTS

We acknowledge the contributions of members of the Cloud Services Labs at Intel Labs Europe, towards the Apex Lake Framework as well as *Michael Mcgrath* for proofreading and providing constructive feedback.

This work is also supported by the Swedish Research Council (VR) through the Cloud Control project (C0590801). Apex Lake builds upon results of collaborative research funded by the EU Projects: SLA@SOI (#216556), Mobile-Cloud Networking (#318109), T-Nova (#619520), RESERVOIR, and IOLanes (#248615).

7. REFERENCES

[1] X. Zhiqun, C. Duan, H. Zhiyuan, and S. Qunying. Emerging of telco cloud. *Communications, China*, 10(6):79–85, 2013.

[2] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS 2012), IEEE*, pages 204–212. IEEE, 2012.

[3] W. Li, J. Tordsson, and E. Elmroth. Virtual machine placement for predictable and time-constrained peak loads. In *Proceedings of the 8th International Workshop on Economics of Grids, Clouds, Systems, and Services (GECON 2011)*, pages 120–134. Lecture Notes of Computing Science, Vol. 7150, Springer-Verlag, 2012.

[4] C. Subramanian, A. Vasan, and A. Sivasubramaniam. Reducing data center power with server consolidation: Approximation and evaluation. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[5] D. Espling, L. Larsson, W. Li, J. Tordsson, and E. Elmroth. Modeling and placement of structured cloud services. *IEEE Transactions on Cloud Computing*, Accepted, 2014.

[6] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[7] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[8] G. Kandiraju, H. Franke, M. Williams, M. Steinder, and S. Black. Software defined infrastructures. *IBM Journal of Research and Development*, 58(2):1–13, 2014.

[9] C. Liu, Y. Mao, J. Van der Merwe, and M. Fernandez. Cloud resource orchestration: A data-centric approach. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, pages 1–8, 2011.

[10] V. Bayon-Molino, M. Marazakis, Y. Klonatos, R. Nou, and J. Giralt. Iolanes deliverable d5.2: Tools for experimental monitoring and logging and tools for analytical evaluation, 2012.

[11] B. Gregg. Use method: Linux performance checklist. `http://www.brendangregg.com/USEmethod/use-linux.html`, September 2013.

[12] A. Edmonds, T. Metsch, A. Papaspyrou, and A. Richardson. Toward an open cloud standard. *Internet Computing, IEEE*, 16(4):15–25, July 2012.

[13] C. Watson, S. Emmons, and B. Gregg. A microscope on microservices. `http://techblog.netflix.com/2015/02/a-microscope-on-microservices.html`, February 2015.

[14] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):4, 2015.

[15] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1):107–117, 1998.

[16] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of the USENIX International Conference on Automated Computing (ICAC 13). San Jose, CA*, 2013.

[17] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *FAST*, pages 163–176, 2011.

[18] H. Brian and R. Roy. Introducing atlas: Netflix's primary telemetry platform. `http://techblog.netflix.com/2014/12/introducing-atlas-netflixs-primary.html`, December 2014.

[19] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 13–24. ACM, 2007.

[20] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*, pages 111–124. ACM, 2010.

III

Paper III

Performance Anomaly Detection using Datacenter Landscape Graphs

Olumuyiwa Ibidunmoye[*], Thijs Metsch[**],
Victor Bayon-Molino[**], Erik Elmroth[*]

[*]*Dept. Computing Science, Umeå University*
SE-901 87 Umeå, Sweden
{*muyi, elmroth*}*@cs.umu.se*
http://www.cs.umu.se/ds

[**]*Intel Labs Europe*
Collinstown Industrial Park,Leixlip, Ireland
{*thijs.metsch, victor.bayon-molino*}*@intel.com*

Abstract: The migration of mission-critical workloads to the cloud and automation of various aspects of datacenter management is contributing to the evolution of software-defined infrastructures. One implication of this evolution is that the composition (both physical and virtual) and logical topology of datacenters is becoming even more dynamic. Identification of performance problems (e.g. bottlenecks) in such environments needs to be aware of this dynamic topology to understand the impact of dependencies among components. We introduce a technique that a) employs expert knowledge to identify bottleneck components using associated performance metrics, and b) uses dynamic dependencies across components to rank problem components. We demonstrated the technique experimentally on an OpenStack testbed with realistic load and bottleneck injection. We observed that the technique is able to detect and rank problem nodes; and may facilitate diagnosis effort.

Performance Anomaly Detection using Datacenter Landscape Graphs

Olumuyiwa Ibidunmoye*, Thijs Metsch†, Victor Bayon-Molino† and Erik Elmroth*
* Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden
Email: {muyi, elmroth}@cs.umu.se
† Intel Labs Europe
Collinstown Industrial Park
Leixlip, Ireland
Email: {thijs.metsch, victor.bayon-molino}@intel.com

***Abstract*—The migration of mission-critical workloads to the cloud and the automation of various aspects of datacenter management is contributing to the evolution of software-defined infrastructures. One implication of this evolution is that the composition (both physical and virtual) and logical topology of datacenters is becoming even more dynamic. Identification of performance problems (e.g. bottlenecks) in such environments needs to be done with awareness of this dynamic topology to understand the impact of dependencies among components. We introduce a technique that a) employs expert knowledge to identify bottleneck components using associated performance metrics, and b) utilizes dynamic dependencies to rank problem components. We demonstrated the technique experimentally on an OpenStack testbed with realistic load and bottleneck injection. We observed that the technique is able to detect and rank problem nodes; and may facilitate diagnosis efforts.**

I. Introduction

Increasing workload requirements, advancements in virtualization and associated cloud computing technologies are contributing to the recent development of software-defined infrastructures (SDIs). Traditional computing resources, such as compute, storage and network are, as a result becoming increasingly software-defined and programmable [1]. Troubleshooting operational problems (e.g. service downtimes, server faults, capacity shortages, etc.) in such environment will be a challenge considering the dynamic nature of the logical topology and composition of the entire computing stack.

Performance troubleshooting in datacenters typically involve monitoring many performance metrics, such as application performance, and resource and energy utilization counters, to determine if hosted services and resources are meeting target service-level objectives (SLO) [2]. Although, analyzing such high-level metrics is indispensable, real-time information about changing dependencies within the datacenter is equally important in order to understand, detect, and circumvent performance issues. This is also essential for enhancing the agility of services and datacenter resources.

Existing datacenter management frameworks and middlewares such as OpenStack[1], Kubernetes[2], and OpenDayLight[3] already provide a growing set of functionalities for managing the large heterogeneous pools of compute, storage and networking resources in a uniform and distributed manner. However, they lack functionalities for tracking and managing topology information dynamically. Also, there is lack of appropriate analytics subsystems to extract insights from real-time operational and topology data to support runtime orchestration for optimized service delivery.

Intel's Apex Lake [3] is a framework for handling numerous orchestration use cases in datacenters especially in SDIs. One major feature of Apex Lake is its concept of *landscapes* which are graph-based models of the temporal datacenter topology. Landscape graphs continuously organize both physical and logical entities (e.g. compute, storage and network resources, and application services) of a datacenter into a multi-dimensional graph structure. Apex Lake combines landscapes with appropriate analytics to discover insights and heuristics to derive datacenter optimizations, such as initial placements, re-balancing, etc. Preliminary deployments of Apex Lake indicated that the ability to quickly discover performance deviations from target SLOs and to identify suspect landscape nodes is an essential use case of the Apex Lake framework. This work is an attempt to address this need. The specific research question is:

> *How to efficiently exploit real-time operational metrics and dynamic spatial relationships in order to identify anomalous landscape entities?*

We propose *landscape colouring*, a problem identification mechanism for classifying anomalous nodes (or entities) in landscape graphs. Anomalous nodes are datacenter components showing manifestation of bottlenecks (e.g. CPU saturation, or crashed disk storage). Landscape colouring leverages expert knowledge to identify anomalous nodes based on associated performance metrics, and utilizes inter-node spatial dependencies to rank multiple anomalous nodes.

[1] https://www.openstack.org/
[2] http://kubernetes.io/
[3] https://www.opendaylight.org/

This paper is organized as follows. This section is concluded with a brief review of related works in Section I-A. Section II introduces Apex Lake. We describe the construction of landscape graphs in Section III. The methodology used by landscape colouring to identify anomalous landscape entities is discussed in Section IV. We describe a short experiment conducted on an OpenStack testbed in Section V to demonstrate our approach. We conclude in sections VI and VII.

A. Related work

Performance troubleshooting research have explored the dependencies among distributed components for problem determination as found by recent survey of the field [4], [5]. This has led to many systems and algorithms including Pinpoint [6], Sherlock [7], LWT [8], Net-Cohort [9], E2EPROF [10], Orion [11], and ADD [12]. However, many of these systems are white-box in nature, and are designed from the application perspective. Hence they typically model only application components without physical resources. Dynamic dependencies across components are mostly derived through source-code instrumentation, tracing application transactions, requests, or call-graphs.

The closest study to this work is VFocus as proposed in [13]. VFocus utilizes graph analytics for troubleshooting and managing datacenters. While VFocus are realized via distributed data-processing graphs (DPG), the graph nodes are data processing agents and may not correspond to an actual datacenter entity. Landscape colouring is based on Apex Lake's landscape graphs whose nodes are directly associated with at least one physical or virtual datacenter entity.

II. Apex-Lake: Smart Orchestration in SDIs

Apex Lake [3] is a framework for supporting smarter orchestrations in cloud datacenters especially in SDIs. It is being actively developed by the Cloud Service Lab of Intel Labs Europe. The main vision of the framework is '*efficiently defining and maintaining a landscape of the physical and logical resources and associated services enriched by end-to-end monitoring, to support setup and run-time orchestration for optimized service delivery*' [3]. The framework provides functionalities that enables various orchestration use cases in SDIs. Orchestration use cases include initial placement decisions (e.g. placement of complex services), re-balancing actuations (e.g. migrating service replica to ensure SLA compliance), and capacity planning (e.g. auto-scaling and hardware upgrades).

Apex Lake consists of three major subsystems that have been identified as required components for smart orchestration. Figure 1 shows the overall architecture of the framework showing these components and the control flows within the context of an SDI. Apex Lake operates according to two control flows, the *foreground* flow and the *background* flow. The foreground flow handles front-facing requests and events while the background flow delivers key insights to make better/smarter decisions via a `Watch-Learn-Decide` cycle.

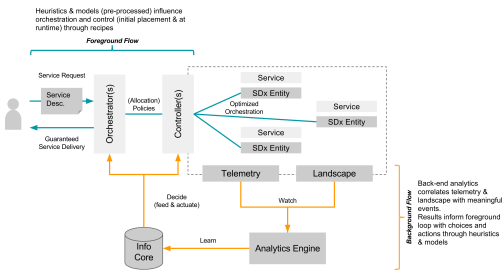


Fig. 1: Overview of Apex Lake showing its major components (Telemetry, Landscape, Analytics Engine, and Info Core) , managed entities (application services; compute, storage, and network resources) and control flows (Foreground flow (green lines), Background flow (orange lines) [3].

Telemetry. *Cimmaron* is Apex Lake's distributed telemetry system for collecting extensive operational indicators about datacenter entities at different granularity (the *watch* step). Examples of operational data include in-band metrics (e.g. service latency, OS kernel counters), out-of-band metrics (e.g. rack controller counters) and environmental metrics (e.g. temperature and energy consumption). Cimmaron's data can be uniformly consumed through its unified data storage layer and can be easily configured to consume data from existing monitoring tools in a publish-subscribe manner. Also it provides a web-based user interface for analysis and visualization.

Landscape. The *landscape* aggregates datacenter into three layers as in Figure 2, representing both the physical topology and spatial connections (or dependencies) between entities in the datacenter at any point in time. There exist a landscape agent responsible for continuously monitoring changes in the spatial relationships and composition of datacenter.

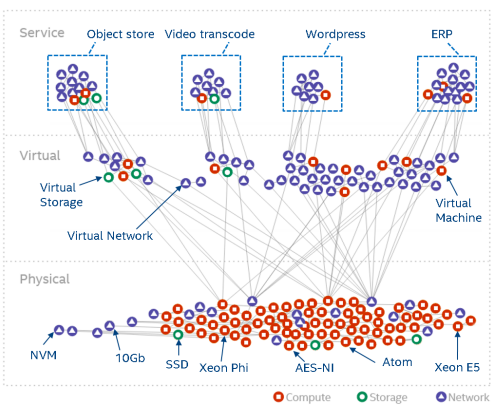


Fig. 2: A snapshot of the aggregation of datacenter entities into landscape graphs; showing the service, virtual, and physical layers as well as the dependency between them [3].

Analytics Engine. This is a PaaS service exposing a set of RESTful API through which the other component (e.g. telemetry and the background flow) consume its functionalities. Aggregated telemetry information are linked to entities in specific scoped landscape graphs and tagged with timestamps to obtain what is known as contextualized *fingerprints*. These fingerprints can be analyzed using data mining and machine learning techniques to derive real-time insights (the *learn* step). The outcome of such analysis can be used for correlating a landscape subgraph (representing a service, a server, or a customer) to the telemetry of its nodes.

Information Core. The *Info Core* is a dynamic repository for managing entity configurations, monitoring data (e.g. availability, utilization, etc.), landscape data, and fingerprints. In general, the Info Core maintains multiple versions of its content in order to capture the temporal nature of the infrastructure. In addition, the background flow can apply machine learning or data mining algorithms on its content to derive *heuristics* and *models* (e.g. decision trees, classification rules, etc.). These heuristics and models are subsequently converted to *recipes* that are used to optimize the behaviour of one or more datacenter entities towards a service objective (the *decide* step). Apex Lake's recipes are machine-readable decision flow (e.g. an auto-scaling or migration procedure) expressed in high-level declarative language.

The current implementation of Apex Lake persists telemetry data collected by Cimmaron in a HBase[4] cluster with query support via an OpenTSDB[5] datastore. Also, landscape graphs are maintained in a Neo4j[6] graph database. RethinkDB[7] is used to store contextual information such as entity behaviour due to scalability issues with Neo4j.

III. Landscape Graphs

The landscape is a graph-based model of the composition of the SDI, the physical topology and spatial relationships (or dependencies) among entities in the datacenter at any point in time. Datacenter entities include servers, server resources, such as compute, memory, storage, and network, as well as components of running application services. The landscape agent in Figure 1 constructs the landscape using live data from OpenStack and organizes the entities into three planes namely; service, virtual and physical layers.

Service Layer. This encompasses all hosted application service components currently running in the datacenter. A service may be composed of web, application and database servers.

Virtual Layer. This layer maps all active software-defined entities in the datacenter such as virtual machines (VM), Linux containers, load balancers, virtual network (e.g. ports, router, etc), and virtual storage (block and object) devices.

Physical Layer. This refers to all physical servers and associated system resources to which virtual entities are mapped. Example of physical entities are servers, RAM, disks, NICs, CPUs, switches, routers, PDUs, etc.

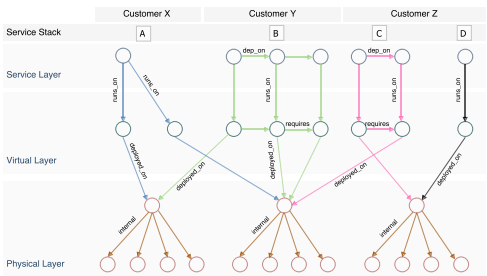


Fig. 3: Schema of a typical landscape graph showing entities (nodes), their dependencies (edges) and specific scopes or subgraphs. Edge label `dep_on` abbreviates `depends_on`.

Figure 3 is a hypothetical landscape of a small datacenter showing the layers, four running service stacks (`A`, `B`, `C` and `D`), the scope of three customers (`X`, `Y`, `Z`), and the dependency across entities. Each landscape node is associated with a set of immutable attributes (e.g. name, type, and layer) and mutable attributes (e.g. VM capacity, floating IP addresses) maintained in the Info Core. Graph edges are semantic relationships or associations between nodes and are directed to indicate intra or inter dependencies. The list and description of available edge labels are presented in Table I. For simplicity, the relationship between a service layer and virtual layer node is directed and one-to-one (e.g. components of an e-commerce service can run in individual VM or together in a single VM). However, the connection between virtual entities and physical entities is many-to-one (e.g. many VMs sharing a server).

The graph structure of the landscape lends itself to interactive visualization and powerful graph analytics in SDIs. For example, subgraphs of specific scopes (or views) can be extracted to support multiple use cases. Based on Figure 3, a subgraph of service `D`, containing 1 service layer node, 1 virtual node, and 5 physical nodes (having 1 server), can be used to optimize its performance. Similarly, a subgraph of Customer `X`, containing 1 service stack with 1 service layer node, 2 virtual nodes, and 10 physical nodes (having 2 servers), can be used to optimize total operating cost. The landscape may also be scoped along or across layers, to optimize datacenter utilization or energy consumption. The landscape agent continuously monitors the SDI for changes due to addition and/or deletion of nodes or edges. Based on this, the background flow of Figure 1 persists in the Info Core versioned fingerprints composed of a given subgraph as well as the mutable attributes and performance metrics of its nodes.

Landscape fingerprints can be analyzed to serve many purposes such as (a) for understanding the topology and composition of services and datacenter components, and tracking how they evolve over time (b) to serve as basis for service or infrastructure orchestration, (c) for identifying problem nodes (e.g. SLO-violation, capacity bottlenecks and

[4]http://hbase.apache.org/
[5]http://opentsdb.net/
[6]http://neo4j.com/
[7]http://rethinkdb.com/

contention) and automatic correction. Section IV of the paper address the last purpose.

IV. Landscape Colouring

Landscape colouring is a mechanism for automatic identification of anomalous nodes in a landscape graph. It addresses the question posed in Section I for a given landscape graph. The concept of 'colouring' is similar to *page colouring* or *cache colouring* when mapping virtual memory to physical memory in operating systems [14]. Page colouring ensures that adjacent or consecutive virtual pages are not mapped to the same physical memory pages which may lead to cache contention. Colouring is done so that physical memory pages with different colours are allocated to different blocks in cache memory. Therefore, to avoid cache conflicts, consecutive virtual pages are mapped to different (but consecutive) colors in physical memory. This scheme thus exploits spatial locality so that conflicts only occur between pages whose virtual addresses differ by a multiple of the cache set size [15].

The idea is that based on real-time performance measurements of landscape entities we can easily classify their behaviour into one of a number of discrete states. The state of a node can then be used to reason if a given node is anomalous or not. Based on spatial relationships with other entities, anomalous nodes can be ranked according to the spatial influence within the given landscape. The assigned states and spatial influence then become some sort of logical colouring that can be used for (a) visualization, (b) for optimizing placement and re-balancing decisions, (c) for identifying performance hot-spots and to 'guide' an orchestrator or a human operator when correcting the hot-spots. In this paper we focus on visualization and identification of anomalous nodes. Given a landscape graph, the landscape colouring technique utilize performance metrics of and spatial relationships among nodes to classify and rank landscape nodes in an online manner.

Landscape colouring rely on core components of Apex Lake to classify nodes of landscapes. Performance metrics (e.g. availability, utilization, saturation, etc.) of each landscape node are provided by Cimmaron, and are versioned, and persisted in the Info Core through the background flow of Figure 1. The landscape colouring processing is also offloaded to the back-end analytics engine.

A. Characterizing landscape nodes

A landscape graph is a directed acyclic graph $g = (V, E), g \subseteq G$, where G is the overall landscape of a datacenter or cluster, V is the set of its vertices or nodes, while E a set of edges. A node, $v \in V$, belongs to at most one layer, and may represent a web server, a virtual machine, or an Intel `Xeon E5` processor. An edge, $e \in E$, is a link indicating the type (Table I) of logical dependency between two landscape nodes. Associated with each virtual and service layer node v is a behaviour tuple $b = (\mathcal{U}, \mathcal{S})$, where $\mathcal{U}$ and $\mathcal{S}$ are each a vector of length k, representing the k most recent values of the node's *utilization* and *saturation* metrics respectively. The metric choice is based on the USE[8] [16] methodology for identifying performance bottlenecks in datacenters.

Utilization. This is the usage of corresponding landscape node (in percentage) in terms of degree to which a resource is busy or what amount of its capacity is consumed.

Saturation. The length of associated resource queue, e.g., CPU run queue, disk requests queue, etc. It captures the quantity of queued work that cannot be serviced given the current capacity of a resource. To keep utilization and saturation metrics uniform, Cimmaron computes saturation as the percentage of time a non-zero queue length was observed over the same interval in which the corresponding utilization value was averaged.

Note that this node definition applies only to virtual and physical layer entities that have a run-time (i.e. deployed within a VM, container, or physical server). The behaviour attribute b of a service node is a k-length vector representing the k most recent observation of the node's application-level performance metric such as service latency or throughput etc.

Technically, not all nodes of the landscape have associated utilization and saturation metrics (e.g. it is only intuitive to talk of the utilization of a server's CPU resource rather than the server itself). Hence, we classify landscape nodes in the physical and virtual layers as either `minor` or `major` entities. Minor entities represent resources *internal* to a virtual or physical server such as CPU, RAM, network, disk storage, etc., while major entities are those with a runtime and contains a number of resources e.g. VM, server, container, or a software-based router running in a VM.

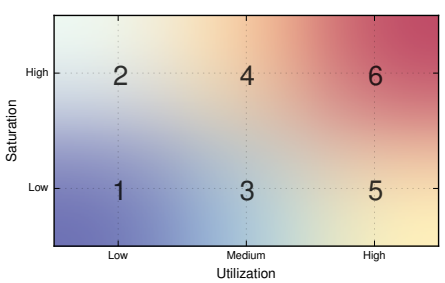


Fig. 4: 2-dimensional map of entity behaviour. The numeric labels indicate six possible states when the utilization and saturation behaviour of a node is plotted on the map.

Figure 4 is a 2-dimensional map showing the possible combination of the two behaviour metrics after they have been discretized into levels. While the saturation of a resource is either `Low` or `High`, its utilization may be `Low`, `Medium`, or `High`. The real-time behaviour measurements

[8]The USE methodology proposes that by analyzing the Utilization, Saturation (e.g. amount extra work waiting on queue), and/or Error (e.g. amount of error events) metrics of individual system resource we can easily characterize capacity bottlenecks at the system-level.

TABLE I: List of landscape edge labels and their semantic meaning.

Label	Description	Type
`depends_on`	Service layer relationship e.g. a web server may depend on a database server	Intra layer
`runs_on`	Service to virtual layer relationship e.g. an application component runs in a virtual layer container (e.g. VM)	Inter layer
`deployed_on`	Virtual to physical layer relationship e.g. a VM deployed on a physical server	Inter layer
`requires`	Virtual layer relationship e.g. a VM requiring a virtual NIC	Intra layer
`internal`	Connections between physical entities e.g. a CPU is internal to a physical server	Intra layer

of landscape entities are characterized by computing the region it falls in on the 2D map. In this paper, the numbered regions are referred to as *states*. Some states have more meaningful interpretations than others depending on the optimization objective. For example, state 2 can be interpreted as an *error* state or failure for a disk exhibiting high queuing rates and an unusually low utilization. Likewise, state 6 may suggests an *over-utilized* landscape entity, such as a fully-utilized CPU with a growing queue. An under-utilized (*cold*) entity is characterized by low queuing and utilization rates. State 5 is a desirable objective in datacenters, we want to achieve high resource utilization with little or no queuing.

Interpretation of regions or states depends on what objective the infrastructure or service provider is trying to optimize. For energy efficiency staying in states 1 or 3 for too long may perhaps be undesirable since the server is kept running while being under-utilized. Migrating jobs from such server to others and shutting it down may potentially improve server and energy utilization on the long run. Hence, for a given goal, we group the states into desirable and undesirable (*anomalous*) states. An entity state is classified anomalous if it remains in undesirable state for most of the time in a observation window.

B. Identifying anomalous nodes

The identification of anomalous landscape nodes are realized through fuzzy classification. The approach is similar to what is proposed in [17]. Fuzzy logic allows for logically reasoning about uncertainty in a given domain in a traceable and interpretable manner. Expert knowledge or heuristics such as the interpretation of states in Section IV-A can easily be formalized in terms of rules and membership functions [18]. In systems literature, hard thresholds are generally used to express vague heuristics such as *high utilization* when CPU utilization is above 70%, for example, and *low utilization* otherwise. Such heuristics generally assume that membership to either levels is binary, i.e., 1 (belong) or 0 (does not belong). Fuzzy logic allows for membership that takes on continuous values within the range [0,1], hence expressing imprecise knowledge such as `Very High` or `Medium` utilization becomes realistic. This accounts for the softness in the boundaries between regions in Fig 4.

The first step in fuzzy classification is fuzzification, a transformation of crisp values of a behaviour pair $b = (\mathcal{U}_t, \mathcal{S}_t)$ into fuzzy inputs. Continuous utilization values are mapped into 3 fuzzy sets, `Low`, `Medium`, and `High`, while saturation values take membership in 2 fuzzy sets, `Low` and `High`. The terms `Low`, `Medium` or `High` are referred to as *linguistic variables*. Elements of a fuzzy set may have varying degrees of membership in the set. Let X be the domain of a metric of interest (e.g. utilization). A fuzzy set A in X is a range of continuous values in X and defined as $A = \{(x, \mu_A(x))|x \in X\}$. The term $\mu_A(x)$ is a membership function $\mu_A : x \rightarrow [0, 1]$ describing degree to which a value x belong in A. Typically, the shape of $\mu_A(x)$ can take different forms such as triangular, trapezoidal, singleton, and bell waveforms [19]. We use the trapezoidal membership functions for the `Low` and `High` fuzzy sets and the bell-shape waveform for `Medium`. Figure 5 shows membership functions of CPU utilization and saturation.

To determine the state of an entity at time given its behaviour pair $b_t = (\mathcal{U}_t, \mathcal{S}_t)$ at time t, we fuzzify $\mathcal{U}_t$ and $\mathcal{S}_t$ according to Fig 5a and Fig 5b respectively. Next we combine the fuzzy inputs using fuzzy rules. A fuzzy rule expresses a given knowledge in terms of linguistic terms as shown in (1). It associates a set of conditions (antecedent) to an output or conclusion (consequent).

$$R_i: \text{ IF } \underbrace{x_1 \text{ is} A_1 \ldots \text{and } x_n \text{ is } A_n}_{\text{rule antecedent}} \text{ THEN } \underbrace{g^* \text{ is } i}_{\text{rule consequent}} . \tag{1}$$

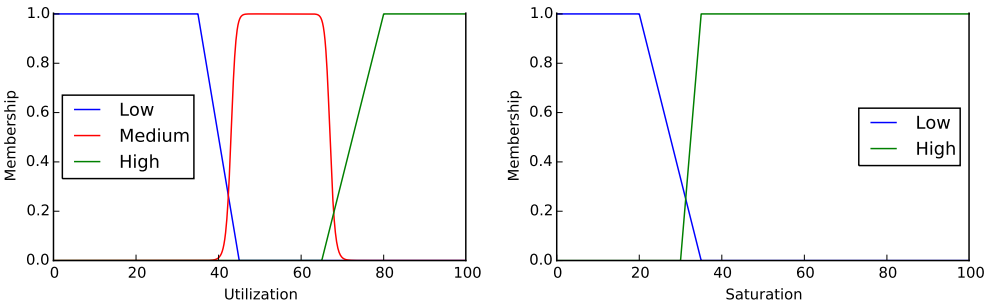
For example, the degree of membership behaviour pair $b_t = (\mathcal{U}_t, \mathcal{S}_t)$ in region 1 is computed using:

$$R_i: \text{IF } Utilization \text{ is } Low \text{ and } Saturation \text{ is } Low \text{ THEN } g^* \text{ is } i \tag{2}$$

where g^* is one of six states in the 2D maps. The membership function of the g^* is a singleton. Fuzzy operators `and` is based on the Mamdani-type fuzzy system. The output of (2) is also called the *support* or *firing strength* for the rule. We generated five more rules in addition to (2) corresponding to the six states in the 2D map. Since an input behavior pair may belong to multiple states at different degrees, we aggregate the supports of all rules using the `Max` method. Let τ be the set of supports for all rules for input $b_t = (\mathcal{U}_t, \mathcal{S}_t)$, then, the state of b_t is the consequent of R_{i^*}, where

$$i^* = \underset{i=1..6}{\arg\max}\, \tau_i$$

Recall that each landscape node is contextualized with behaviour $b = (\mathcal{U}, \mathcal{S})$ where $\mathcal{U}$ and $\mathcal{S}$ are vectors containing k most recent observations of the two metrics in the current window. We classify each observation using the procedure above, the state of the entity for the window is determined by majority vote. A node is anomalous if its derived state belongs to the set of undesirable states. The states and classes of nodes are then used to *colour* the landscape by appending them to the mutable attributes of corresponding nodes.



(a) Membership function of CPU `utilization`

(b) Membership function of CPU `saturation`

Fig. 5: Conversion of crisp metric values for CPU `utilization` (a) and `saturation` (b) metrics to fuzzy sets using trapezoidal and bell-shape membership functions. The bell membership function of the *Medium* fuzzy set in (a) is parameterized by mean $\mu = 55$ and standard deviation $\sigma = 14$.

C. Ranking anomalies

Due to the size of landscape graphs, many anomalous nodes may be identified at the same time which may easily overwhelm administrator, alert management system, or the orchestrator. Also, some landscape entities are more important than others in terms of the number of upper layer nodes that depend on them. Intuitively, such nodes should be given priority when identified as anomalous. Similarly, due to long-range dependencies and correlated faults in datacenters, anomalous nodes having many higher-level anomalous nodes may be given more priority than others. Hence there is a need to prioritize the list of anomalous entities not only in terms of their influence but also in terms of the impact on SLOs. To quantify the influence of individual entities in the landscape, we exploit the inherent dependency structure in the landscape to guess correlations and dependencies within and across layers.

We define an influence rank using the PageRank [20] algorithm, to estimate how important a node is within a landscape. PageRank is a link analysis algorithm used by Google Search to quantify the relative quality of websites. A website is said to be of high quality if it receives links from important websites. Hence, a landscape node assumes higher influence rank if it receives links from important nodes. Given a coloured landscape digraph, $G' = (V, E)$, the influence rank, `InfRank`, of an entity v is computed as

$$\text{InfRank}(v) = (1 - \alpha) + \alpha \sum_{v' \in V'} \frac{\text{InfRank}(v')}{\text{outdeg}(v')} \qquad (3)$$

where $v' \in V'$ is a set containing all upper layer entities that directly depend on v. With outdeg(v'), each entity spreads its influence out evenly among all entities they depend on. The share of influence contributed by v' to v is given by $\sum_{v' \in V'} \frac{\text{InfRank}(v')}{\text{outdeg}(v')}$. The individual influence contributed by each entity depending on v is *damped down* by the parameter $0 < \alpha < 1$. The $(1 - \alpha)$ term makes up for entities in the leaves of the physical layer having no out-going links.

Furthermore, to quantify the impact of anomalous entities, we define an impact rank, `ImpRank`, which is derived based on the influence rank as follows:

$$\text{ImpRank}(v) = \text{InfRank}(v)(1 + \beta\lambda + n) \qquad (4)$$

where β is a weight for an entity being anomalous or not while λ is a weight a entity being from the physical, virtual or service layers of the landscape and n is the number of upper layer nodes that are also anomalous and is computed dynamically. Parameter β of an anomalous entity is set to 1 and to very small value otherwise (e.g. 0.01). Also, λ is set so as to express the importance of the layers. For example, $(\lambda_{\text{physical}} = 0.6,\ \lambda_{\text{virtual}} = 0.2,\ \lambda_{\text{service}} = 0.1,)$, expresses that anomalous nodes in the physical layer should be prioritized than the other layers. While the influence rank already captures spatial dependencies, parameter n, is a means to propagate correlated anomalies in the landscape, such that anomalous physical layer entities with greater number of anomalous virtual layer entities will have higher impact rank. This approach can be used to prioritize resource nodes depending on workload mix. With an additional parameter we can indicate how resource nodes should be prioritized, such as ranking storage entities higher under an I/O-intensive workload in the second.

Finally, anomalous landscape entities are sorted according to `ImpRank` in descending order. High `ImpRank` implies an important anomalous entity that should be attended quickly otherwise may results in SLO violations in the immediate future. Such priority list can then be followed to issue alarm emails to human operators, serve as recommendation for remediation by automated management systems such as Apex Lake's orchestrator. In order to rectify observed anomalies, the orchestrator may take scaling or rebalancing decisions and trigger actuation actions to realize required optimization.

D. The Algorithm

Given a landscape graph g contextualized with telemetry from the recent k-length window, Algorithm 1 applies the above procedure to identify and rank anomalous entities in g based on a set of parameters described in Table II.

The algorithm proceeds in two passes. In the first pass, we compute the influence rank of all nodes in g using the PageRank algorithm, we visit all minor nodes in the physical and virtual layers, classify their behaviour using the fuzzy classifier, and compute the associated impact ranks. In the second pass, we update the rank score of anomaly nodes previously identified in the first pass. This is done by counting the number of anomalous minor nodes in the virtual layer that is directly dependent on the the predecessor of an anomalous minor node in the physical layer. The output is a set of anomalous entities ranked according to their impact score and classification of each node.

The implementation of Algorithm 1 is realized using `NetworkX`[9] which provide support for attributed digraphs, fast graph manipulation and efficient computation of various measures such as degrees, density, hub and authority scores, and page ranks. The fuzzification and inference operations are implemented using modules provided by `scikit-fuzzy`[10].

V. Evaluation

In this section, we present two experimental case studies to demonstrate the effectiveness of our landscape colouring technique. We examine the ability to identify anomalous nodes under a strictly compute-intensive workload in the first scenario, and under a mix of compute-intensive and I/O-intensive workloads. The experiments were performed on an Apex-Lake-enabled OpenStack testbed.

The setup is an OpenStack testbed composed of 2 HP Proliant physical servers across which 5 VMs running 2 services are distributed as shown in Fig 6. The capacity of the physical servers include; 8-core Intel(R) Xeon(R) CPU E5320 1.86 GHz processor, 7 GB RAM, 60 GB RAID disks and Gigabit Ethernet. Each VM is configured with 4 virtual CPUs, 20 GB storage, and 1 GB memory, running Ubuntu 14.04. The original landscape of the testbed consists nearly a hundred nodes. For clarity, we scaled the landscape to show only the major nodes in Figure 6 while the landscape used in our demonstration contains both minor and major nodes as in Figure 11. We deploy two application stacks, A and B, with similar workloads running on VMs, (vm_1, vm_2, vm_3), and (vm_4, vm_5), respectively. While vm_1, vm_2, and vm_5 are deployed on physical machines pm_1, vm_3 and vm_4 are mapped to pm_2. This static landscape is used through out the experiments, we leave the dynamical landscapes case for future work.

Case 1: We show that our landscape colouring approach is able to identify and rank correctly anomalous compute nodes. The experiment involves putting load on the system so as to inject compute contentions in the testbed. We deployed `stress`[11] to emulate CPU-intensive job in each VM by spawning a number of workers executing `sqrt()` in C. The duration of the experiment is for 1 hour, which is further divided into four 15-minute periods. In each period we configured `stress` to induce some VMs to consume more CPU than available. The mapping of VM to period is:
$W_1 \leftarrow vm_1, vm_4, pm_2, pm_1$
$W_2 \leftarrow vm_1, vm_3, vm_4, vm_5, pm_1$
$W_3 \leftarrow vm_3, vm_4, pm_1$

[9]NetworkX is a Python library for graph creation, manipulation and processing. Available: http://networkx.github.io/

[10]http://pythonhosted.org/scikit-fuzzy/

[11]http://people.seas.harvard.edu/~apw/stress/

Algorithm 1: Landscape colouring and ranking

Input: $g = (V, E)$, b, α, λ, β, s^*
Output: A*, C
Initialization
$A \leftarrow [\,]$
$C \leftarrow [\,]$
Identification pass:
InfRank $\leftarrow (1-\alpha) + \alpha \sum_{v' \in V'} \frac{\text{InfRank}(v')}{\text{outdeg}(v')}$
for $v \in V$ **do**
  $c \leftarrow FALSE$
  if v ***is*** $minor$ ***and*** $(physical$ ***or*** $virtual)$ **then**
    $\vec{u}, \vec{s} \leftarrow b[v]$
    $s \leftarrow fuzzyclassifier(\vec{u}, \vec{s})$
    if $s \in s^*$ **then**
      $c \leftarrow TRUE$
      $A \leftarrow append(A, v)$
    end
  end
  $C[v] \leftarrow c$
  let $l \leftarrow$ layer of v
  ImpRank$[v] \leftarrow$ InfRank$[v](1 + \beta[c] \cdot \lambda[l])$
end
Rank update pass:
for $v^* \in A$ **do**
  $m \leftarrow 0$
  if v^* ***is*** $minor$ ***and*** $physical$ **then**
    $P_{v^*} \leftarrow predecessor(v^*)$
    $V_{v^*} \leftarrow predecessor(P_{v^*})$
    for $n \in V_{v^*}$ **do**
      $v^*_{suc} \leftarrow successor(n)$
      for $v' \in v^*_{suc}$ **do**
        if v' ***is*** $minor$ ***and*** $C[v']$ **then**
          $m \leftarrow m + 1$
        end
      end
    end
  end
  ImpRank$[v^*] \leftarrow$ ImpRank$[v^*] +$ InfRank$[v^*] \cdot m$
end
$A^* \leftarrow sort\_descending(A, \text{ImpRank})$
return A^*, C

TABLE II: Description of parameters

Parameters	Description
g	A landscape graph, a service or scoped subgraph
α	Damping factor for computing PageRank
β	A vector of weights for normal and anomalous entities
λ	A vector of weights associated with each layer
b	A vector of pairs of entity behaviour $(\vec{u}, \vec{s})$
$s*$	A set of undesirable states derived from Figure 4
InfRank	Influence ranks of nodes
ImpRank	Impact ranks of nodes
A	List of anomalous nodes
$A*$	List of anomalous nodes after ranking and sorting
C	Dictionary of nodes' classifications
m	Number of anomalous virtual layer nodes dependent on an anomalous physical node

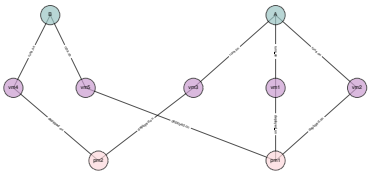


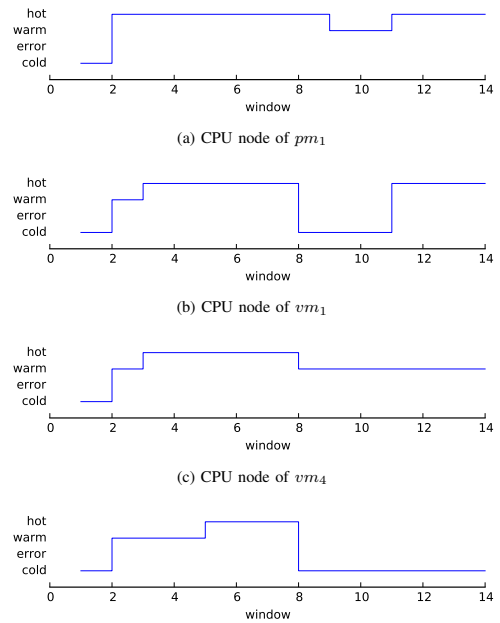
Fig. 6: Experiment setup with 2 servers

$W_4 \leftarrow vm_1, vm_3, vm_2, pm_1$
Given this schedule we expect that contention for CPU resources will occur in one or more nodes; and should CPU node of pm_2, for example, be anomalous, it should be ranked higher than other CPU nodes, since many virtual CPUs depend on it.

Cimmaron reports telemetry (utilization and saturation metrics) about each entity per second which is further aggregated at 1 minute level to smooth out spikes. Identification of anomalies is performed at a 5-minute interval window. At each time window, the landscape graph g is contextualized using the metric values within that window and serves as input to Algorithm 1. Parameters of the algorithms are defined as follows: $\alpha = 0.85$, $\beta_{\text{normal}} = 0.01$, $\beta_{\text{anomaly}} = 1$, $\lambda_{\text{physical}} = 0.6$, $\lambda_{\text{virtual}} = 0.3$, $\lambda_{\text{service}} = 0.1$. The detection objective in this case is to detect and rank entities showing signs of *over-utilization* for at least 3-minute in the 5-minute window. Hence, using the 2D map in Figure 4 we designate region 6 (*hot* state), as well as 2 and 4 (*error* states) as undesirable anomalous state. Based on our objective, regions 1 (*cold*), 3 (*cool*) and 5 (*warm*) will be desirable. It is easy to see that when the objective is to *minimize energy consumption* for example, then states 1 and 3 will unlikely be desirable. It would be better to offload jobs from an under-utilized server to others for example.

Figure 12 is the graph obtained at the end of period W_1 with compute node of vm_1, vm_4, vm_5, and pm_1 highlighted as anomalous. This landscape corresponds to window 6 of Figure 7 showing the state evolution of culprit nodes as step functions[12]. The state of an entity according to a majority vote is plotted against each 5-minute window. It is clear that the algorithm correctly identifies the over-utilized nodes corresponding in window 6. While vm_1, vm_3, vm_5, and pm_1 are expectedly anomalous, vm_3 however is not anomalous. This may be due to VCPU scheduling effects in host pm_2. Since both vm_3 and vm_4 are deployed on the same host with total CPU demand greater than physically available, vm_4 may get more CPU time than vm_3 due to *stolen* CPU cycles which effectively starves vm_3 [21]. According to our ranking strategy, CPU node of pm_1 is ranked higher than others in Figure 8 because it is a physical entity and has 2 VCPUs dependent on it. The ranking intuitively enforce a differentiation of entities. Anomalous nodes are those that will result in most impact on SLA and SLO compliance.

[12]Only the step functions of anomalous nodes highlighted in Figure 12 are plotted.



(a) CPU node of pm_1

(b) CPU node of vm_1

(c) CPU node of vm_4

(d) CPU node of vm_5

Fig. 7: `Case 1`: Step functions showing how the behaviour of each compute node evolved over time.

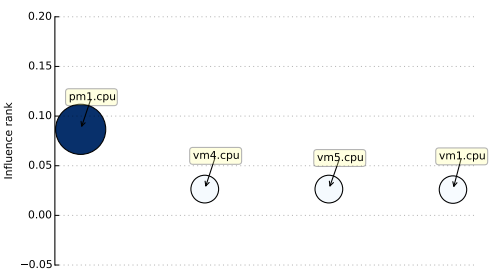


Fig. 8: `Case 1`: Impact ranks of anomalous nodes. The size of each cycle corresponds to the rank score of associated entity. Entities with the same colors have the same rank score.

Case 2: The objective remain the same as the first case except that we deal with a mixed workload (CPU and disk IO). We used `stress` to generate CPU load as in `Case 1` and also to emulate IO jobs by spawning 1 disk worker writing megabytes of disk pages without disk caching in each VM. The duration, parameters and window size remain the same. In addition to the schedule of CPU load in `Case 1`, VMs with induced IO load are mapped to periods as follows:

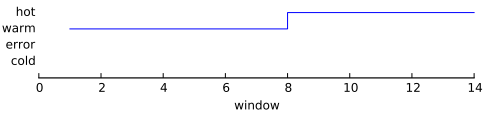
$W_1 \leftarrow vm_3$ (200 MB), vm_5 (50 MB)
$W_2 \leftarrow vm_1$ (100 MB), vm_2 (100 MB)
$W_3 \leftarrow vm_3$ (50 MB), vm_4 (100 MB), vm_5 (200 MB)
$W_4 \leftarrow vm_3$ (100 MB)
Using this complex setup we expect to induce both compute and disk I/O contention that at some point. According to Figure 13, our algorithm identified 7 anomalous nodes including 2 storage entities at the last period corresponding to windows 10-13 in Figure 9[13]. Notice that the occurrence of anomalous events is not entirely deterministic. For example, while the storage entity of vm_3 is clearly anomalous during the first five windows (Figure 9c), but constantly writing 100 MB disk pages in windows 10-13 (period 4) does not seem to saturate the disk I/O bandwidth enough. The storage node of pm_1 is anomalous at this time due to the spatial dependency between vm_5 and pm_1 and temporal dependency between consecutive periods and windows. Note that the storage node of vm_5 was injected with a 200 MB IO load together with vm_3 in the previous period. The combined IO load perhaps results in queuing of disk requests on pm_1 which makes its storage node to be fully utilized in the fourth period. We show step functions of only the interesting anomalous entities.

VI. Discussion

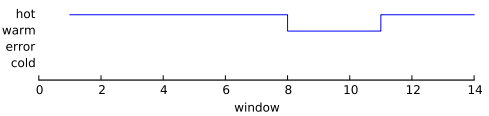
Landscape colouring is a completely automatic process that can be used as a starting point for root-cause analysis of performance issues in the datacenter. By continuously monitoring and colouring the landscape we can capture the prevailing states of entities, quantify and rank their implications on neighboring entities and the landscape as a whole. Such insights can then be used to enrich multiple decision points. For instance, it could prevent impending a service performance violation before it happens by raising alerts once some physical or virtual layer entities of a service stack start showing signs of problems. In a large datacenter with many resources and services, the ranking scores could give an indication of what order corrective actions be taken. This is not only useful for automated remediation of problems, it could refine and save operators'efforts. Though it is primarily proposed as an anomaly detection use case for Apex Lake, landscape colouring can also be used to support other management operations (e.g. initial placement, re-balancing, etc.). For example, Apex Lake's could use insights from coloured landscapes to guide an orchestrator to steer workloads away from physical layer entities exhibiting intermittent anomalous behaviour.

The fuzzy membership functions (in Figure 5) use predefined thresholds to define the range of values for the fuzzy sets. The values used in our experiments are obtained from literature and recommendation by system administrators. However, these thresholds are highly contextual and depend on what customers use the datacenter, which workloads they run (storage or network bound), and what objectives the datacenter operator is trying to optimize for (e.g. low CPU utilization for mobile applications vs higher CPU utilization for HPC jobs). Ideally the system should learn these values automatically to reflect changes in workloads

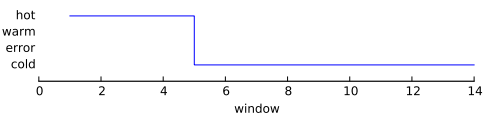
[13]Only the step functions of anomalous nodes highlighted in Figure 13 are plotted.



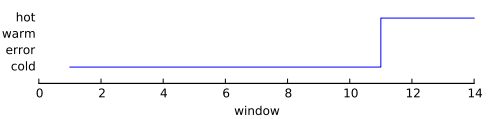
(a) CPU node of vm_3



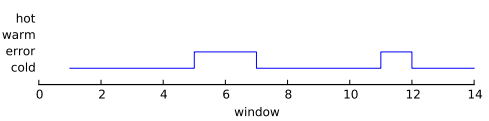
(b) CPU node of pm_1



(c) Storage node of vm_3



(d) Storage node of vm_5



(e) Storage node of pm_1

Fig. 9: `Case 2`: Step functions showing how the behaviour of each compute node evolved over time.

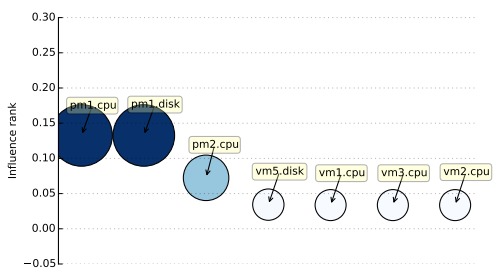


Fig. 10: `Case 2`: Impact ranks of anomalous nodes at window 6. The size of each cycle corresponds to the rank score of associated entity. Entities with the same colors have the same rank score.

and service objectives. Though the temporal dependency between consecutive entity behaviours is not explored, this property could be exploited for predicting future entity

states.

To improve the quality of detection and ranking it is desirable to integrate service-level anomaly correlation with landscape colouring. This will allows us to correlate application-level anomaly events with infrastructure-level bottlenecks so that alarms are raised for only relevant anomalies that result in actual SLA violations. Presently, we are working on testing our approach on a larger testbed running real applications and one where edges between landscape nodes actually vary over time in order to evaluate its scalability and accuracy. Subsequently, how the technique could be used to trigger corrective actions automatically and to enrich other datacenter optimizations within the Apex Lake framework will be explored.

VII. Conclusion

We presented the design and implementation of landscape colouring, a mechanism for automatic identification of anomalous nodes in landscape graphs of datacenter entities. Our algorithm combines telemetry data about individual nodes, spatial dependencies across landscapes, and goal-specific heuristics to characterize prevailing behaviour of entities and detect nodes with suspicious behaviour. To enhance automatic corrective actions and to minimize load on human operators, we ranked anomalous entities according to their impact on the neighbouring nodes and on specific service objectives. Through experimental case studies on an OpenStack testbed, we have demonstrated that our mechanism is able to detect and rank problem nodes. In the future, we plan to conduct more experiments on a larger testbed to evaluate the scalability of the technique.

VIII. Acknowledgments

We acknowledge the contributions of members of the Cloud Services Labs at Intel Labs Europe, towards the Apex Lake Framework especially Surya Narayanan Natarajan for helping with the experiments. This work is supported by the Swedish Research Council (VR) through the Cloud Control project (C0590801).

References

[1] C. Li, B. Brech, S. Crowder, D. M. Dias, H. Franke, M. Hogstrom, D. Lindquist, G. Pacifici, S. Pappe, B. Rajaraman *et al.*, "Software Defined Environments: An Introduction," *IBM Journal of Research and Development*, vol. 58, no. 2/3, pp. 1–1, 2014.

[2] L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[3] T. Metsch, O. Ibidunmoye, V. Bayon-Molino, J. Butler, F. Hernández-Rodriguez, and E. Elmroth, "Apex Lake: A Framework for Enabling Smart Orchestration," in *Proceedings of the Industrial Track of the 16th International Middleware Conference*, ser. Middleware Industry '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:7. [Online]. Available: http://doi.acm.org/10.1145/2830013.2830016

[4] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance Anomaly Detection and Bottleneck Identification," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 4:1–4:35, Jul 2015. [Online]. Available: http://doi.acm.org/10.1145/2791120

[5] C. Wang, S. P. Kavulya, J. Tan, L. Hu, M. Kutare, M. Kasick, K. Schwan, P. Narasimhan, and R. Gandhi, "Performance Troubleshooting in Data Centers: An Annotated Bibliography?" *ACM SIGOPS Operating Systems Review*, vol. 47, no. 3, pp. 50–62, 2013.

[6] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," in *In Proceedings of International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 595–604.

[7] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies," in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4. ACM, 2007, pp. 13–24.

[8] R. Apte, L. Hu, K. Schwan, and A. Ghosh, "Look Who's Talking: Discovering Dependencies between Virtual Machines Using CPU Utilization," pp. 17–17, 2010.

[9] L. Hu, K. Schwan, A. Gulati, J. Zhang, and C. Wang, "Net-cohort: Detecting and Managing VM Ensembles in Virtualized Data Centers," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 3–12.

[10] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham, "E2EProf: Automated End-to-end Performance Management for Enterprise Systems," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2007, pp. 749–758.

[11] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions." in *OSDI*, vol. 8, 2008, pp. 117–130.

[12] A. Brown, G. Kar, and A. Keller, "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment," in *IEEE/IFIP International Symposium on Integrated Network Management Proceedings*. IEEE, 2001, pp. 377–390.

[13] C. Wang, K. Schwan, B. Laub, M. Kesavan, and A. Gavrilovska, "Exploring Graph Analytics for Cloud Troubleshooting," in *11th International Conference on Autonomic Computing (ICAC 14)*, 2014, pp. 65–71.

[14] R. E. Kessler and M. D. Hill, "Page Placement Algorithms for Large Real-indexed Caches," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 338–359, 1992.

[15] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam, "Compiler-directed Page Coloring for Multiprocessors," in *ACM SIGPLAN Notices*, vol. 31, no. 9. ACM, 1996, pp. 244–255.

[16] B. Gregg, *Systems Performance: Enterprise and the Cloud*. Pearson Education, 2013.

[17] S. Cateni, V. Colla, and M. Vannucci, "A Fuzzy Logic-based Method for Outliers Detection." in *Artificial Intelligence and Applications*, 2007, pp. 605–610.

[18] J. A. Roubos, M. Setnes, and J. Abonyi, "Learning Fuzzy Classification Rules from Labeled Data," *IEEE Trans. Fuzzy Systems*, vol. 8, no. 5, pp. 509–522, 2001.

[19] Y. Bai and D. Wang, "Fundamentals of Fuzzy Logic Control—Fuzzy Sets, Fuzzy Rules and Defuzzifications," in *Advanced Fuzzy Logic Technologies in Industrial Applications*. Springer, 2006, pp. 17–36.

[20] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1, pp. 107–117, 1998.

[21] R. van Riel, "Measuring Resource Demand on Linux," in *Linux Symposium*, 2006, p. 287.

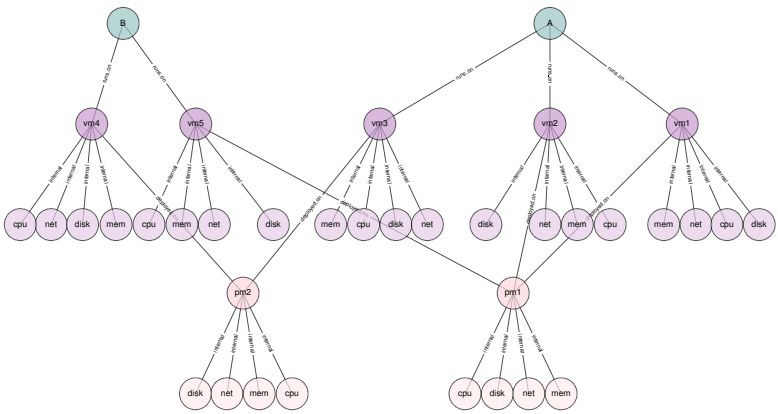


Fig. 11: Initial landscape graph. A node represent an application, a VM, a resource (e.g. CPU), or physical server. Colors: service layer nodes are in green, virtual entities in purple, while physical entities are in orange.

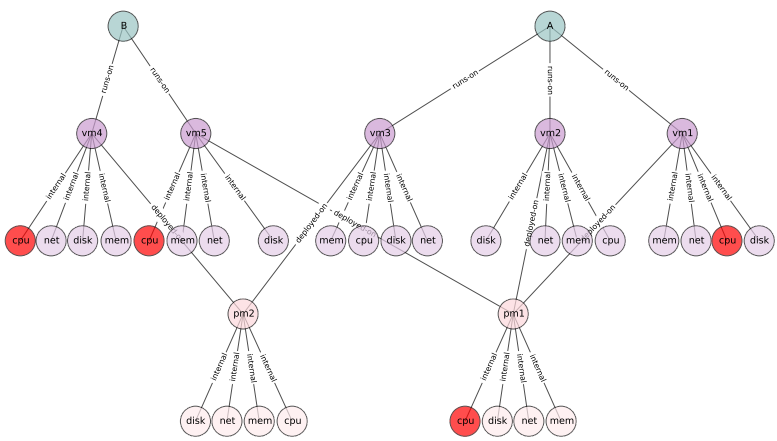


Fig. 12: `Case 1`: Coloured landscape showing anomalous compute entities (highlighted in red) at window 6.

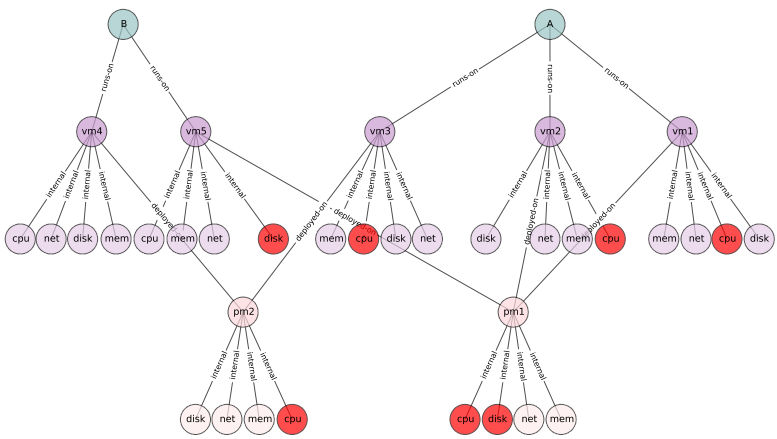


Fig. 13: `Case 2`: Coloured landscape showing anomalous compute and storage entities (highlighted in red) at window.

IV

Paper IV

Real-time Detection of Performance Anomalies for Cloud Services

Olumuyiwa Ibidunmoye[*], Thijs Metsch[**], Erik Elmroth[*]

[*]*Dept. Computing Science, Umeå University*
SE-901 87 Umeå, Sweden
{*muyi, elmroth*}*@cs.umu.se*
http://www.cs.umu.se/ds

[**]*Intel Labs Europe*
Collinstown Industrial Park, Leixlip, Ireland
{*thijs.metsch*}*@intel.com*

Abstract: In order to prevent violation of service-level objectives and to guarantee good user experience, detection of symptoms such as slow application response, degraded transaction throughput, and service outages, is crucial. We propose a black-box approach for detecting such symptoms in service performance behaviour without intrusive application instrumentation. In case a known baseline behaviour exists, we employ kernel density estimation to discover deviations from a given set of baseline measurements. Conversely, when no baseline exists, we apply statistical process control charts on prediction errors obtained from Holt-Winter's double exponential smoothing to identify anomalies in metric time-series. We evaluate our methods on tail response times traces collected from experiments conducted in a real testbed under realistic load and fault injections. Results show the applicability of our approach for improving service assurance and also demonstrate how service level anomalies correlate with system-level events such as resource contention and bottlenecks.

Real-time Detection of Performance Anomalies for Cloud Services

Olumuyiwa Ibidunmoye
Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden
Email: muyi@cs.umu.se

Thijs Metsch
Intel Labs Europe
Collinstown Industrial Park
Leixlip, Ireland
thijs.metsch@intel.com

Erik Elmroth
Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden
Email: elmroth@cs.umu.se

***Abstract*—In order to prevent violation of service-level objectives and to guarantee good user experience, detection of symptoms such as slow application response, degraded transaction throughput, and service outages, is crucial. We propose a black-box approach for detecting such symptoms in service performance behaviour without intrusive application instrumentation. In case a known baseline behaviour exists, we employ kernel density estimation to discover deviations from a given set of baseline measurements. Conversely, when no baseline exists, we apply statistical process control charts on prediction errors obtained from Holt-Winter's double exponential smoothing to identify anomalies in metric time-series. We evaluate our methods on tail response times traces collected from experiments conducted in a real testbed under realistic load and fault injections. Results show the applicability of our approach for improving service assurance and also demonstrate how service level anomalies correlate with system-level events such as resource contention and bottlenecks.**

I. Introduction

Modern application services are characterized by complex architectures and unpredictable load traffic (such as load spikes and flash-crowd[1] behaviour). These factors often explain performance anomalies such as slow application response, degraded transaction throughput, and outages in cloud-based application services today. Also, the nature of the underlying virtualized infrastructure in which most services are hosted affects service performance in many ways. Cloud infrastructure platforms, such as Amazon EC2, package and provision computational resources in virtual machines (VMs) to numerous service providers in an on-demand manner. Though the initial capital expenditure is greatly minimized for service providers, the co-location of heterogeneous services from multiple providers is sometimes a source of concern for service owners.

According to our previous survey [1], cloud service performance anomalies are often the manifestation of many problems ranging from application-level bugs, workload spikes, and correlated systems faults (e.g. resource contention). Fig. 1 is schematic diagram of the cause-effect relationships between issues from disparate sources and how they together induce capacity bottlenecks and eventually anomalies in service performance.

[1]Internet phenomenon where a network suddenly receives a huge influx of traffic due to incidents such as breaking news, natural disasters, demise of famous persons, etc.

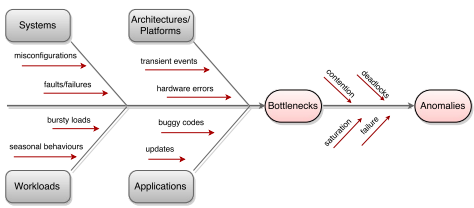


Fig. 1: Causes of performance anomalies in systems (source: [1]).

1) *Applications*: Application-level issues such as incorrect parameter settings, buggy codes, and software updates may induce unexpected performance behaviour [2].
2) *Workload*: Bursty workloads, such as caused by flash-crowds, and change in seasonal load patterns may induce undesirable effects such as congested queues, oversubscribed threading resources, and eventually intermittent or sustained spikes in performance measurements.
3) *Architectures and Platforms*: Sometimes transient errors or events in the underlying architecture or operating system–such as the JVM & Intel SpeedStep technology [3], may introduce hard-to-detect transient bottlenecks in systems.
4) *Systems*: System-level issues such as misconfiguration, faulty components, or capacity bottlenecks may induce performance anomalies in many ways. In cloud environments, one major culprit is the inherent performance interference experienced when applications with similar resource needs and workload patterns are co-located on a single server.

Due to the complexity of the runtime environment, diagnosing service performance issues has become a major task in the life-cycle of virtualized services. Service providers generally want to be able to detect unwanted behaviour quickly and in real-time. In our survey [1], we observed that, while there exist many studies focusing on the detection of anomalies in multi-dimensional system-level performance metrics (as done in [4] [5] and [6]), studies on detecting application performance anomalies are few. Existing approaches for detecting anomalies in application

performance metrics (e.g. response time and throughput) either resort to simplistic mechanisms based on thresholding (such as in [7], [8], and [9]) or require intrusive application instrumentation (such as tracing request flows in [10] and [11]).

In this paper, we focus on achieving robust anomaly detection in application performance metrics (APM) under two scenarios. In the first scenario, we assume there exist a baseline profile–a collection of *known* or *typical* metric values collected over a period of stable service workload and configuration. The task is to detect deviations from the given baseline. This is informed by trends in the fields of network [12] and database [13] management, where deviations in real-time traffic measurements from known baseline (typical) behaviour are considered as anomalies [14]. In the second scenario, the notion of typical behaviour is unknown a priori and we must detect deviations in prevailing service behaviour. We phrase these two cases in the following questions:

Q1: *How to detect deviations in real-time service performance measurements based on a given baseline profile?*

Q2: *How to detect deviations in real-time service performance measurements when a baseline is a priori unknown ?*

To address *Q1*, we perform a behaviour-based anomaly detection by exploiting the probability distribution of the given baseline profile. A non-parametric kernel density estimation (KDE) technique is used to detect point anomalies in real-time APM measurements. Given that there exist no baseline in case of *Q2*, we perform prediction-based anomaly detection instead. The prediction-based approach performs a one-step ahead forecast of metric value, tracks the residual errors and flags points where statistical properties (such as mean shift) of the residuals change abruptly. Forecasting is done using trend-preserving double exponential filtering while exponentially weighted moving average (EWMA) control chart is used to detect mean shifts.

In Section II, we layout details of the behaviour-based approach using the KDE model while Section III describes how the forecasting and control chart methods are constructed for predictive anomaly detection. We describe the experimental approach for evaluating the techniques in Section IV. Evaluations and results are presented in Section V. We conclude with some discussions, description of related works and conclusion in sections VI, VII, and VIII respectively.

II. Behaviour-based anomaly detection

Traditionally, APM values are said to be anomalous when they violate a reference point (i.e. threshold). For example, suppose $X = \{x_i\}_{i=1}^{n}$ is a set of n observations of a given performance metric, a threshold-based anomaly detection scheme may define a threshold $\mathcal{T}$ as $\mathcal{T} = k\sigma_X + \mu_X$ where μ_X and σ_X are the mean and standard deviation of values in X respectively and k is a multiplicative factor of the standard deviation. An observations $x_i \in X$ is classified anomalous if it satisfies the condition, $x_i \geq \mathcal{T}$. The idea behind this threshold is that normal values of the performance metric are expected to be within k standard deviations (σ_X) about the mean μ_X. There are some obvious limitations to this approach. First, the assumption is that the distribution of APM values will be Gaussian (i.e. normally distributed) and therefore will result in greater false alarms on data with arbitrary or non-Gaussian distribution. Secondly, thresholds are hard to appropriately specify and may soon become outdated due to changing runtime environment and workload patterns. It is also not suitable for scenarios where the reference point is a set of baseline measurements rather than a scalar value. Such [13].

In this section we present a non-parametric approach (similar to [15]) to address the aforementioned issues with scalar thresholds. Given a set of baseline measurements, our method estimates a model of the unknown probability distribution of the baseline using Kernel Density Estimation (KDE). We subsequently classify new observations (such as real-time metric measurements) based on the conditional densities estimated by the KDE model. However, the standard KDE algorithm is known to produce spurious density approximations due to the use of global bandwidths. We implement an adaptive KDE following the procedure in [16] by incorporating local bandwidths to account for local variations in the data. Due to the time complexity of the KDE we also implement both the standard and adaptive KDE using k-dimensional trees [17] and compare their performance later in Section V.

A. Kernel Density Estimation (KDE)

The basic idea of the behaviour-based anomaly detection has to do with exploiting the *probability density function* (PDF) or *density* of the given baseline values. The PDF or density of a continuous random variable is a function that relates the relative likelihood for the random variable to assume a given value in a population [18]. Given comparable capacity configurations and workload characteristics, metric values obtained from a running application should reveal similar metric distribution as the baseline from the same application under normal conditions. Significant deviations from the baseline distribution may indicate an anomaly. Some existing studies have proposed similar approaches in performance studies such as in [19] and [20].

The KDE is a proximity-based technique for detecting novel patterns in data with the assumption that normal instances are far more frequent than anomalous instances [21]. Given a univariate set of metric values $X : \mathbb{R}^{n\times 1}$, the unknown density function $\hat{f}(X)$ can be estimated by aggregating a set of scaled kernel functions as follows;

$$\hat{f}(x) = \frac{1}{nh}\sum_{i=1}^{n} K_h(\frac{x - x_i}{h}) \tag{1}$$

Each kernel function $K_h(\cdot)$ is centered at each data point with width controlled by a *bandwidth* parameter h. The bandwidth h, acts as a smoothing factor controlling the variance from one point to another in the data. Since h remains the same for all kernels, it is also referred to as a *global bandwidth*. The scaled kernel $K_h(\cdot)$ of the form $\frac{1}{h}K(\frac{\cdot}{h})$ is any symmetric, function that satisfies $\int_{-\infty}^{\infty} K_h(\cdot)d(\cdot) = 1$

and $K_h(\cdot) \geq 0$ [22]. We have used a set of Gaussian kernels for computing the density estimate $\hat{f}(x)$ according to Eq. 1. Though other standard kernels such as the Tophat, Epanechnikov, Exponential, etc., exist, the Gaussian kernel is most commonly used in density estimation due to its ability to produce smooth density estimates [23]. A Gaussian kernel is defined as [21]:

$$K_h(y) = \frac{1}{h\sqrt{2\pi}} \exp(-\frac{y^2}{2h^2}) \tag{2}$$

The choice of kernel $K_h(\cdot)$ does not significantly affect the accuracy of the estimate as much as the bandwidth parameter h. High values of h produce very smooth estimates but may ignore local variance while low values produce less smooth but potentially more accurate estimates [15]. The value of h may be set to an arbitrary value, to a plug-in values such as the AMISE and Silverman's approximations [24], or to values derived through grid-search cross-validation.

B. Adaptive KDE (AKDE)

The use of a fixed global bandwidth, h in Eq.1 ignores local variations in density from one data point to another [21]. We follow the procedure in [16] to adapt the amount of smoothing based on the local density in the data (i.e. the bandwidth, h, is allowed to vary from one data point to another). First, a pilot density $\hat{f}^*(x)$, is estimated from the entire dataset using Eq. (1), such that $\hat{f}^*(x_i) > 0, \forall i$. Then individual local density is computed using the square root rule as, $\lambda_i = \{\frac{\hat{f}^*(x_i)}{g}\}^{-\alpha}$ where g is the geometric mean of $\hat{f}^*(x_i)$ i.e. $\log(g) = n^{-1} \sum_{i=1}^{n} \log(\hat{f}^*(x_i))$. The adaptive KDE estimate becomes:

$$\hat{f}(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{h\lambda_i} K_h(\frac{x - x_i}{h\lambda_i}) \tag{3}$$

The term $h\lambda_i$ now acts as the local bandwidths such that the width of the kernel placed at each x_i is equal to $h\lambda_i$. Setting $\alpha = 0.5$ is generally recommended as it offers good estimate in most cases [16]. Equation (3) is equivalent to (1) when $\alpha = 0$ since $\lambda_i = 1$. Note that the same value of h used to obtain the pilot, $\hat{f}^*(x_i)$, must be used in computing Eq. (3) so as to ensure that the actual estimate is sensitive to the same scale as the pilot.

C. Computational issues with density estimation

The KDE belongs to a class of statistical learning problems called the *generalized n-body problems* with the complexity of a naive solution being $\mathcal{O}(n^2)$. Riegel et al [25] has shown that problems in this category can be efficiently computed in $\mathcal{O}(n\ log\ n)$ using specialized data structures such as the k-dimensional trees [17]. A KD-tree is a spatial data structure for organizing data points in k-dimensional space and readily offer $\mathcal{O}(n \log n)$ k-nearest neighbour computations. The implementation of the KDE models is based on the idea that if a data point $x_0 \in X$ is geometrically distant from a set of points $\vec{y} \subset X$, then it is sufficient to compute the density estimate of x_0 only from the set $X - \vec{y}$, i.e., the k-nearest neighbours of x_0. To implement the standard and adaptive KDE models we used the `KD-Tree` class in `scikit-learn` [26].

D. Detecting anomalies based on the density estimation

Given a baseline profile $\mathcal{B}$ containing a set $X = \{x_i\}_{i=1}^{n}$ of performance metric values. First we learn a model, $\theta_{\mathcal{B}}$ corresponding to the kernel density estimate, $\hat{f}_{\mathcal{B}}(X)$, of the baseline values. To classify an observed data point, x_0, we compute a conditional density $p(x_0|\theta_{\mathcal{B}})$–the likelihood of X taking on the value of x_0 given its probability distribution $\theta_{\mathcal{B}}$. For a suitably chosen parameter, ϵ, x_0 is classified anomalous if $p(x_0|\theta_{\mathcal{B}}) < \epsilon$ implying that x_0 is not likely to have been generated by $\theta_{\mathcal{B}}$. Samples with relatively small $p(x_0|\theta_{\mathcal{B}})$ are considered anomalous and those for which $p(x_0|\theta_{\mathcal{B}})$ is high lie in dense regions and thus are normal data.

III. Prediction-based anomaly detection

The approach in Section II is based on training a model to recognize new independent data that do not belong to distribution of typical or normal data. In some use cases, it maybe difficult to obtain a baseline or may quickly become obsolete. The challenge is to monitor the service in order to detect unknown events or unexpected changes in service performance behaviour that may suggests workload spikes, system-level faults or resource bottlenecks due to contention among co-located services. The prediction-based anomaly detection exploits the temporal dependency in time series of successive APM measurements to detect abrupt changes in the underlying statistical property of the data (e.g. mean shift). The assumption is that such property is expected to exhibit little or no variation from one time step to another. Consistent shift of a certain magnitude may suggest a change in the time series behaviour and hence an anomaly. In this section we present an online mechanism for achieving predictive anomaly detection using a combination of Holt-Winters forecasting and EWMA control charts.

A. Residual generation using Holt-Winters forecasting

We consider a sequence of APM measurements as a time series of the form $X = \{x_1, \cdots, x_{t-1}, x_t, x_{t+1} \cdots\}$. Exponential smoothing is a common method in statistical forecasting literature and have been successfully used in diverse domains of applications. The basic idea is to forecast future values using weighted averages of past observations, with weights decaying exponentially over time [27]. Holt-Winters [28] forecasting extends basic exponential smoothing to account for trends and seasonality in a time series data. Holt-Winters is based on the assumption that observed time series can be decomposed into three components: a *level* component (the intercept of a signal), ℓ_t, a linear *trend* (the long term direction or slope), b_t, and a *seasonal* (the systematic, calendar related movement) component, s_t. Studies such as in [29] has shown the basic component form of the Holt-Winters model to not produce good estimates for the level and seasonal components. Their work suggests methods for correcting estimates produced by the method. According to Hyndman et al in [27], the additive and error correction form of the smoothing equations is given as:

$$\ell_t = \ell_{t-1} + b_{t-1} + \alpha e_t \tag{4}$$

$$b_t = b_{t-1} + \alpha\beta e_t \tag{5}$$

$$s_t = s_{t-m} + \gamma e_t \tag{6}$$

where α, β and γ are smoothing parameters for the level, trend, and seasonal components respectively and are set to values in the range $[0, 1]$. The index of seasonality, m, denotes the length of a season (e.g. $m = 12$ for monthly data with yearly trend). The one-step-ahead forecast at time t using previous forecast of the level, trend and seasonality at time $t-1$ is thus $\hat{x}_{t|t-1} = \ell_{t-1} + b_{t-1} + s_{t-m}$. The corresponding forecast error or residual, e_t, is computed as $e_t = x_t - \hat{x}_{t|t-1}$, where x_t is the observed value at time t and $\hat{x}_{t|t-1}$ is the predicted value using the most recent smoothed components from time $t-1$. The statistical property of the time series of residuals, $E = \{e_1, \cdots, e_{t-1}, e_t, e_{t+1} \cdots\}$, will be exploited in the next section for anomaly detection.

The Mean Absolute Percentage Error (MAPE) is a commonly used measure of forecast accuracy [27]. The MAPE is defined as $MAPE = \frac{1}{n}\sum_{t=1}^{n} \frac{|y_t - \hat{y}_t}{y_t}| \times 100$, where y_t is the observation at time t and $\hat{y}_t$ is the forecast of y_t. The optimal settings for parameters α, β, and γ are those that minimize the MAPE. We will use this strategy to determine optimal parameter settings in Section V.

B. Detecting deviations using EWMA control chart

Statistical process control (SPC) is a collection of tools widely used to achieve stability and monitor variability in production processes [18]. A control chart is a graphical display showing individual values of a quality characteristic, Y, against the sample number or time. The chart contains a center line (CL), the mean of Y under normal conditions. The upper control limit (UCL) and lower control limits (LCL) are horizontal lines drawn above and below the CL respectively [18]. Generally, control charts operate under the premises that output of the monitored process are normally distributed and so appropriates values for the LCL and UCL can be determined from a normal bell curve. Therefore they are not suitable to be applied directly on time series of APM measurements with unknown distribution, trend, or seasonal variation. Instead we apply control charts on forecast residuals obtained in Section III-A.

The most commonly used control charts are the Shewhart's $\bar{x}$, s, and R control charts. However, these charts are not suitable for APM measurements as they either assume normality or that measurements arrive in batches. We employ the Exponentially Weighted Moving Average (EWMA) control chart. EWMA control charts have been shown in studies [30] to be robust against non-normality and are particularly suitable for case where measurements arrive sequentially rather than in batches [18]. Given forecast residuals $e_1, \cdots, e_{t-1}, e_t, e_{t+1} \cdots$. We construct an EWMA chart of the form:

$$e'_t = \lambda e_t + (1-\lambda) e'_{t-1} \tag{7}$$

Equation (7) exponentially smooth the forecast residuals over time controlled by parameter λ ($0 < \lambda \leq 1$). The center line and control limits for the EWMA control charts is then defined as follows [18]:

$$(UCL, LCL) = \mu_0 \pm L\sigma\sqrt{\frac{\lambda}{(2-\lambda)}\left[1-(1-\lambda)^{2t}\right]} \tag{8}$$

where λ and L are design parameters for the EWMA charts, while λ controls the rate of smoothing, L is a multiple of the standard deviation and controls how sensitive the chart is to shifts around the center line. Parameter L, is generally set to 3 in literature [31]. Suppose the residuals e_i are independent random variables with mean $\mu_0 = \frac{1}{T}\sum_{t=1}^{T} e_t$, the CL, and with variance σ^2, then the variance of each e'_i is given as $\sigma^2_{e'_i} = \sigma^2 \frac{\lambda}{(2-\lambda)}\left[1-(1-\lambda)^{2t}\right]$.

The limitation of this formulation is the assumption that entire time series measurements are available so that we can easily estimate parameters μ_0 and σ a priori before setting up the control chart. This is not always possible when dealing with dynamic service performance data with storage constraints. Variation in service workload may cause the center line and in turn the upper and lower limits, to shift over time. To overcome this issue we modify Eq. 8 so that parameters μ and σ are computed adaptively as measurements arrive using the Welford algorithm [32]. The Welford technique iteratively updates the mean and variance online using data observed up to each time step. The implication of this is that the control limits UCL and LCL will also vary along with the center line and variance. However, as t grows the term $[1-(1-\lambda)^{2t}]$ approaches unity and the control limits will converge to asymptotic (steady-state) limits $\mu_0 \pm L\sigma\sqrt{\frac{\lambda}{(2-\lambda)}}$ [18].

Observations that violate the control limits are in statistical literature said to be *out-of-control*. Such observation are what we call anomalies since they are indications of changes in the time series behaviour due to load spikes, bottlenecks, bugs or faults. To detect anomalies, we monitor consecutive values of the smoothed forecast errors (e'_t), and out-of-control observations are classified as anomalies. Specifically, out-of-control observations are those for which the predicates $e'_t > UCL$ or $e'_t < LCL$ hold.

A general measure of the effectiveness of control charts is the *Average Run Length* (ARL). The ARL is the average number of points that must be plotted before a point indicates an out-of-control condition [18]. For EWMA control charts there exists a trade-off between the number of out-of-control detection and the ARL based on the value of parameter λ. Lower values of λ yield lower ARL and in turn higher of out-of-control anomalies. Conversely, when λ is close to unity a high ARL is observed and as a result fewer anomalies are recorded. In general, λ should be high enough so as to reduce the false alarm rate and to ensure greater sensitivity to anomalies.

We describe data specific issues such as parameter initialization in Section V when we evaluate the algorithms on specific datasets.

IV. Experimental Design

The goal of our experiments is to evaluate the two anomaly detection schemes (in sections II and III) on real performance traces obtained from a benchmark web application under realistic load and fault injections deployed in a virtualized environment. In this section we describe the experimental testbed, the test application, workload emulation and the experimental methodology.

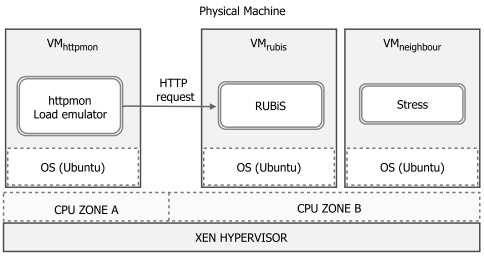


Fig. 2: Experimental testbed.

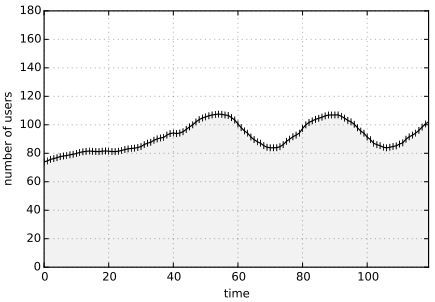


Fig. 4: Workload intensity

A. The Testbed

The experiments were conducted on a HP ProLiant physical machine (PM) equipped with a total of 32 CPU cores[2], 56 GB of memory, four 500G SATA disks in RAID10 and Gigabyte Ethernet. The server is virtualized using the Xen Hypervisor [33]. The testbed (in Fig. 2) consists 3 virtual machines, $VM_{httpmon}$ running the load emulator, VM_{rubis} (configured with 16 VCPUs, 4GB memory and 25GB storage capacity), hosting RUbiS benchmark application, and an antagonist, $VM_{neighbour}$ (6 VCPUs, 4GB memory and 25GB storage), used to inject faults into the environment. We deploy VM_{rubis} and $VM_{httpmon}$ on the same PM with compute-level isolation in order to reduce network delays. Compute isolation is achieved by partitioning physical CPUs into two non-overlapping pools, `A` (10 cores) and `B` (22 cores) using Xen's CPUpools policies[3]. Each pool may have entirely different scheduler and VMs are assigned to pools on creation and can be moved from one pool to another. $VM_{httpmon}$ is allocated to `Zone A` while the other two are allocated to `Zone B`.

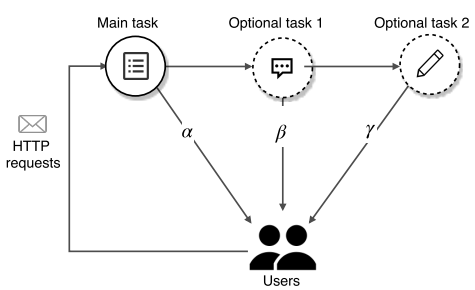


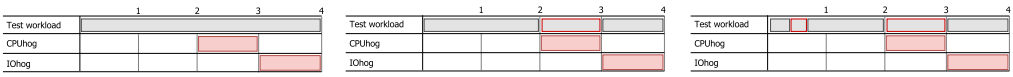
Fig. 3: A model of workload mix generated by `httpmon`

1) Benchmark Applications: The benchmark web application used in our experiments is *RUBiS*[4]. RUBiS is an eBay-like web-based e-commerce application that provides selling, browsing and bidding functionalities. The PHP variant of RUBiS is deployed in VM_{rubis}. The VM is running `Apache 2.0` as its web server (with module `Apache MPM prefork` enabled for thread safety and request isolation), and `MySQL 5.0` as the database server. The `MaxClients`, `ServerLimit` parameters for `Apache` have been set to relatively high values to accommodate high load. Also, to introduce compute and IO perturbations into the environment, we used the open-source `stress`[5] tool deployed in $VM_{neighbour}$.

2) Workload and Performance metric: Fig. 3 is a simple model of how the workload mix used in our experiments are is generated. It shows the pattern of interactions between emulated clients and RUBiS. The interaction consist of a dynamic set of users concurrently sending HTTP requests to a main task such as fetching a random product page. With probability α, they exit the application or proceed to perform an optional task (e.g. requesting all past comments about the product) with probability β and/or perform another optional task (e.g. submitting a remark about the product) with probability γ. End-users are emulated using, `httpmon`[6], a versatile HTTP (`GET` or `POST`) traffic generator. We adapted `httpmon` to support the use of arbitrary workload intensity (Fig. 4) and mix (Fig. 3). The `thinktime` parameter of `httpmon` is set to `900`ms in `OPEN` loop mode for all experiments, while the `concurrency` is held constant for one minute per observation from the workload intensity profile. This technique allows us to stage realistic workload and application behaviour.

The performance of the benchmark application is assessed by the Response Time (RT) or latency of individual HTTP (`GET`/`POST`) request. RT is the time elapsed from when the first byte of a request is sent to the time the last byte of its response is received. `HTTPMON` measures request latencies and aggregates by taking the 95th percentile latency of all requests observed over a given time interval. The choice of tail (95th percentile) response time (P_{95}-RT) is because (a) RT metric are commonly used to reason about user satisfaction and quality of experience in Internet services (b) 95th percentile aggregation enables consistent latency measurements [34] and correlates more with violation of service-level objectives (SLO) than averages (which is easily skewed by spikes).

[2]Two 2.1GHz AMD Opteron™ 6272 processors, with 16 cores each.
[3]http://wiki.xenproject.org/wiki/Cpupools_Howto
[4]http://rubis.ow2.org/index.html
[5]http://people.seas.harvard.edu/~apw/stress/
[6]https://github.com/cloud-control/httpmon



(a) Timeline of experiment 1. (b) Timeline of experiment 2. (c) Timeline of experiment 3.

Fig. 5: Ghant charts showing the timeline of experiments and the injection of anomaly-inducing events. Gray boxes in red border correspond to periods where workload spikes have been injected into the test workload while the red corresponds to injection of CPU and IO contention.

3) Experiment Process: The first step is to generate the baseline used for behaviour-based anomaly detection in subsequent experiments. This is accomplished by running the benchmark (VM*rubis*) for about an hour without VM*neighbour* in the setup in order to eliminate unwanted noise. We injected the synthetic workload intensity and mix according to Fig. 4 and Fig. 3 respectively. The workload mix probabilities are initialized as $\alpha = 0.60, \beta = 0.35, \gamma = 0.05$. We randomly sample the P_{95}-RT over consecutive three-minutes windows to obtain 21 baseline values. The distribution of values in the baseline is shown in Fig. 6. The green region is shown for mere aesthetic since negative latency values are unreasonable. In the evaluation section, we will describe how the SLO-violation annotation is used to quantify the accuracy of the KDE models. We perform

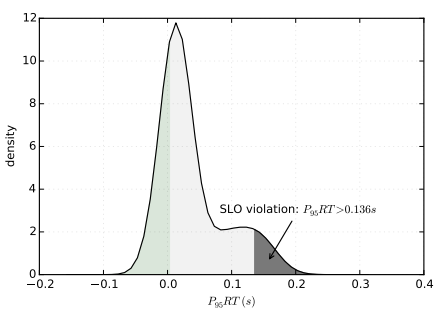


Fig. 6: Distribution of the baseline P_{95}-RT.

three similar experiments to emulate how load spikes and resource contention due to VM co-location cause anomalies in service-level performance metric behaviour. Each experiment runs for two hours following the same workload profile as the baseline experiment. The P_{95}-RT of the application is sampled at 20 seconds interval. A Ghant chart describing the timeline and composition of the testbed as well as the injection of anomaly-inducing events for each experiment is shown in Fig. 5. The entire duration is divided into four 30-minutes long periods. The `test workload` event refers to running VM*rubis* while the `CPUhog` and `IOhog` events refer to running VM*neighbour* to emulate compute and IO contention in the execution environment. For compute contention, we configured `stress` so that it aggressively consumes as much CPU as available in `Zone B`. Also, `stress` is used to induce IO contention by writing blocks of bytes into the disk without disk caching. In addition, we injected workload spikes (by arbitrarily increasing the number of users) between period two to three in experiment 2 and 3, as well as within period one and two in experiment 2.

TABLE I: Description of acronyms and symbols

Acronym / Symbol	Method	**Description**
KDE, AKDE	Behaviour-based	Standard and adaptive kernel density estimation
KDETree, AKDETree	Behaviour-based	k-dimensional tree variants of KDE and AKDE
EWMA	Prediction-based	Exponentially weighted moving average control chart
ARL	Prediction-based	Average run length of the control chart
CL, UCL, LCL	Prediction-based	Center line, upper and lower control limits
τ	Behaviour-based	SLO violation threshold (Section V-A)
h	Behaviour-based	The global kernel bandwidth (Eq. 1 & 3)
ϵ	Behaviour-based	The conditional density threshold (Section II-D)
α, β	Prediction-based	Holt-Winter's smoothing parameters (Eq. 4 - 6)
ℓ_t, b_t	Prediction-based	Time series level and trend components (Eq. 4 - 6)
λ	Prediction-based	Smoothing parameter for EWMA chart (Eq. 7)
L, σ	Prediction-based	Control limit parameters (Eq. 8)
e_t, e'_t	Prediction-based	Forecast and smoothed errors (Eq. 7)

V. Evaluation

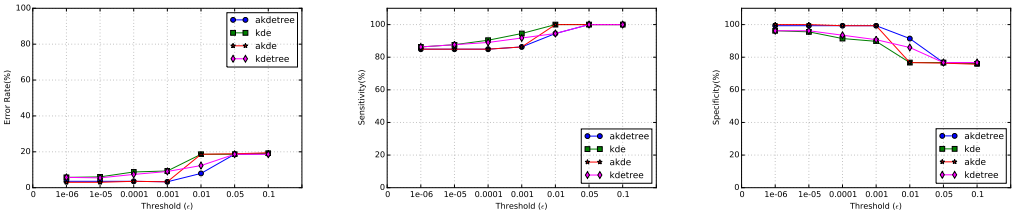
In this section, we evaluate the proposed anomaly detection schemes based on three experiments described in the previous section. To enhance readability, we describe acronyms and symbols used so far in Table I.

A. Behaviour-based anomaly detection

The KDE models described in this paper and the raw data from our experiments are unsupervised in nature, hence it is not so straightforward to quantify the accuracy of the models. We follow the approach in supervised anomaly detection where a label is provided for each observation in the dataset distinguishing which ones are anomalous and which are not [21]. To label our data, we define a threshold τ equal to the 95$^{\text{th}}$ percentile of the values in the baseline (see Fig. 6). The dataset is then labeled according to the following rule:

$$y_i = \begin{cases} 0, & \text{if } x_i < \tau, \\ 1, & x_i \geq \tau. \end{cases} \tag{9}$$

x_i and y_i are the value and class label of observation i respectively. Observations with $y_i = 0$ are normal (below the SLO violation threshold) while $y_i = 1$ are anomalies (above the SLO violation threshold). Fitting a KDE model as described in Section II requires setting global bandwidth h to an optimal value. It also requires choosing the conditional density threshold ϵ that guarantees good model accuracy. To select the right parameters we tried different combinations of h and ϵ. We used grid search cross-validation to optimize the value of h which resulted in $h = 10$, a rather high value compared to the Silverman's [23] approximation, $h = \frac{3n}{4}^{-\frac{1}{5}} \approx 0.57$ (recall $n = 21$). We tried different h within the range $(0.5, 10)$ while varying ϵ as well. The



(a) Misclassification rate (b) Sensitivity (c) Specificity

Fig. 7: Performance of the KDE models for behaviour-based anomaly detection in terms of (a) the overall misclassification rate, (b) ability to correctly identify anomalies, and (c) ability to correctly distinguish normal observations.

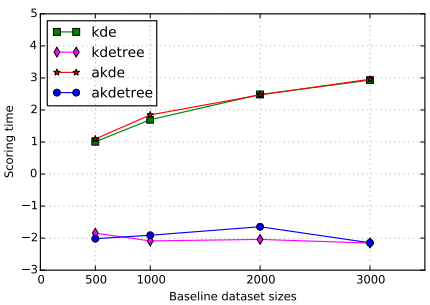


Fig. 8: Time requirement of KDE models at varying test data sizes (in log scale).

observation is that higher bandwidths ($h > 1.5$) result in over-fitting leading the model to mis-classify anomalies as normal observations.

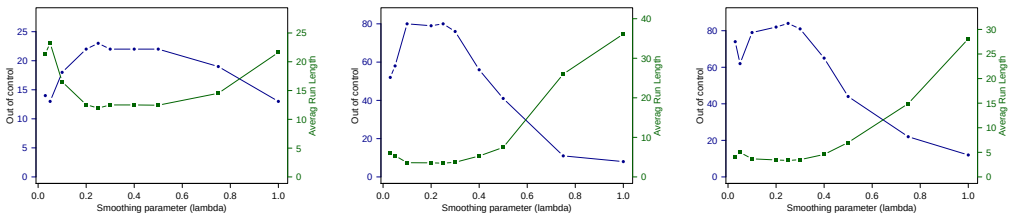
Fig. 7 shows the performance of the KDE models as well as their tree implementations for $h = 1.5$ with varying ϵ while Fig. 7a is the misclassification rate of the model. Fig. 7b is the represents the percentage of actual anomalies in the test dataset that are correctly identified and Fig. 7c is the percentage of normal data instances that are correctly identified by the model. It can be observed that the AKDE model and the tree variants offer consistently low error rates at thresholds below 0.01 after which only the tree variants remain low. Though Fig. 7b indicates that the tree variants are not able to correctly detect anomalies as much as the non-tree variants at lower thresholds, Fig. 7c shows that they are however better at identifying normal data even at higher thresholds. For the given baseline, a threshold of 0.01 seems to offer the best trade-off between model accuracy and sensitivity to anomalies. At this threshold, the tree implementation of AKDE is the preferred model because it offers the best trade-off between mode accuracy and ability to correctly classify both anomalies and normal samples. Furthermore, we fit the models on data obtained from experiments 2 and 3 with parameter settings $h = 1.5$ and $\epsilon = 0.01$. The tree-based AKDE model also performed better than the rest at error rates of 8% and 14%, and sensitivity of 95% and 99% for experiments 2 and 3 respectively.

Fig. 8 confirms the $\mathcal{O}(n^2)$ complexity of the standard KDE model under varying test data sizes. The plot illustrate the number of seconds required (in log scale) to completely compute the conditional density of observations in test datasets of varying sizes. With their $\mathcal{O}(n \log n)$ time complexity, the tree variants offer greater speed.The tree models thus become scalable and faster as the data size increase in comparison to the standard implementations. However, the AKDE model demands additional time requirement for computing local density bandwidth using the pilot estimate which can get somewhat expensive for large baseline data.

B. Prediction-based anomaly detection

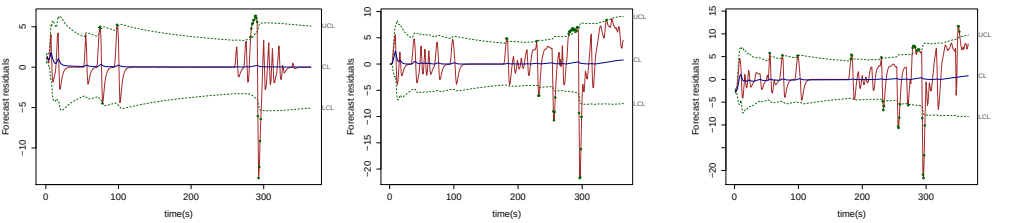
In this section, we use the same datasets from experiments 1, 2 and 3 as in Fig. 5 to evaluate the capability of the predictive anomaly detection scheme to detect anomalies in service performance. The most important part of this is parameterizing the Holt-Winter's forecasting algorithm and the EWMA control chart. The Holt-Winter's forecasting scheme described in Section III incorporates seasonality in the data through the s_t term of Eq. 6, controlled by parameter γ. Since datasets from our experiments do not exhibit any seasonality, we set γ to zero. This reduces the problem to only finding the optimal values for α and β needed to control the time series level and trend respectively. Recall that lower values of α gives more importance to observations in the past than the recent. The same applies to β, values close to zero means that trends in the distant past affect the current forecast than the recent trends. We searched over a grid of possible combinations of α and β, and for each pair we fit the Holt-Winter's model and select the pair with low mean absolute percentage error (MAPE). Hence, we obtained $\alpha = 0.9$ and $\beta = 0.3$ for the Holt-Winter's forecasts. To initialize Holt-Winter's forecasting we set $\ell_0 = x_1$ and $b_0 = x_2 - x_1$, where x_1 and x_2 are the first and second observation of the time series respectfully.

For each test data, the Holt-Winter's method is used to generate forecast residuals on which the EWMA control chart is applied. The parameter L of Eq. 8 controls how wide the control limits are and in turn the sensitivity to shifts in the behaviour of measurements. We set L to the value of 3 (i.e. 3σ) as commonly done in literature [18] [30]. Equation 7 requires that parameter λ is set appropriately to control both



(a) Experiment 1 (b) Experiment 2 (c) Experiment 3

Fig. 9: Predictive anomaly detection: Plots showing how the number of anomalies detected and the Average Run Length (ARL) vary with respect to λ at $L = 3$.



(a) Experiment 1 ($\lambda = 0.81$) (b) Experiment 2 ($\lambda = 0.6$) (c) Experiment 3 ($\lambda = 0.61$)

Fig. 10: Predictive anomaly detection: Adaptive EWMA control chart applied to smoothed Holt-Winter's forecast errors using $\alpha = 0.9$, and $\beta = 0.3$. The indicated control limits are obtained using $L = 3$ (i.e. 3σ). Anomalies are the out-of-control points in green.

the rate at which out-of-control alarms are raised and the average run length (ARL). In Fig. 9 we illustrate the trade-off between number of out-of-control (anomalies) detection and the ARL when λ is varied between 0 and 1. Control charts with higher ARL are generally known to have fewer out-of-control detections and false alarms. Hence, λ should be chosen to yield lower ARL while sensitive enough to shifts of varying magnitude in the time series behaviour. Using Fig. 9, we set the value of λ to 0.81, 0.6, and 0.61 for experiments 1, 2 and 3 respectively. These values correspond to the intersection of the growing ARL line and the decaying out-of-control line that partitions the plots into two regions; a region of low ARL and high out-of-control detection and a region of high ARL but low out-of-control detection. To initialize the EWMA control chart, we set $e'_0 = e_1$, CL to the mean of the first $k = 3$ residuals and σ^2 to the variance of this mean as described in Section III.

Fig. 10 are the control charts obtained when the adaptive EWMA control chart scheme is applied to Holt-Winter's forecast residuals from the experiments. Notice how the control limits (UCL, CL and LCL) follows the trend and shift in the signal. The out-of-control points (i.e. anomalies) are the indicated green points on the plots. The three control charts clearly detect more anomalies at points corresponding to between periods 2 and 4 respectively of the time-line in Fig. 5, indicated as a period of unstable shifts in the time series around the center line. The workload spikes injected in experiments 2 and 3 does not seem to induce as much shift in the data as resource contention between periods 2 and 4. Hence, fewer anomalies are detected within that period. This behaviour suggests that sometimes low-level events such as resource saturation may not readily manifest as anomalies in service performance. By manual observation of the pages accessed by requests to VM_{rubis} and its system metrics, most of the spikes between period 2 and 4 in the charts is due to CPU saturation and database reads and writes that coincide with IO noise from $\text{VM}_{neighbour}$. In addition, spikes in the beginning of the charts are due to initial disk activity before the disk cache is sufficiently warmed up and the initial phase through which the control charts adjusts to the steady behaviour of the time series.

VI. Discussions

The behaviour-based scheme is useful for two scenarios (a) when a baseline profile is available and the task is to identify anomalies in previously unseen APM measurements and (b) when a baseline is not explicitly provided but the entire set of APM dataset is and the task is to identify anomalous values in the given data off-line. The first scenario, then the algorithm described in Section II is applicable directly. In the second scenario, the task is to identify anomalous measurements from the entire data off-line. The behaviour-based technique is still applicable by

using the entire data as both the baseline and test data. First we fit the AKDE model on the data and an appropriate value is determined for the density threshold ϵ. In one more pass over the data, the model is used to classify each observation as anomaly or normal. The prediction-based approach on the hand is best for the second scenario and most suitable when measurements arrive sequentially in a pre-defined interval.

In comparison, the EWMA control chart technique generally detects fewer anomalies and false alarms than the KDE models due to its adaptive nature. The exponential smoothing of the residuals means that spurious spikes are smoothed out and alarms are only raised for sustained anomalies. The KDE model however detects more anomalies accurately than the control chart scheme some of which are false alarms. To reduce occasional spurious alarms due to spiky data, on method is to define a hanging window within which we set a threshold on the maximum number of consecutive anomalies. For example, let $\mathcal{A}$, be the number of sequential anomalies detected within a k-length window, $\mathcal{W}$, an alarm is raised when $\mathcal{A} > \mathcal{L}$, where $\mathcal{L}$ is the maximum number of permissible consecutive anomalies.

For longer experiment periods, we would expect the performance of the KDE to worsen in time as the baseline becomes obsolete. The technique can however be adapted for use in an on-line manner like the predictive scheme by taking recent windows of observations as the baseline and using the trained model to predict in the next window. The classification threshold ϵ can also be set to reflect changing SLO thresholds. However, deciding whether to use the tree variants of the standard KDE or adaptive KDE will depend on the trade-off between speed and accuracy of detection. On one hand, the adaptive KDE will yield fast $\mathcal{O}(n \log n)$ complexity but require significant time for computing local bandwidths through pilot density estimation especially for large baseline data. On the other hand, the tree implementation of the KDE will offer fast classification without prior pilot estimate at the expense of accuracy.

Though the prediction-based technique is easily amenable to real time deployment, it does require getting the parameter settings right. Also the fact that out-of-control samples are also included in computing the control limits may also cause the control limits to be too wide at certain points that it masks actual anomalies. Moreover, the fact that few to no anomalies were generated in some periods (e.g. between period 1 to 2) of experiment 3 is interesting. It suggests that because a service has not yet exhibited anomalous behaviour does not mean there are no suspicious events at the system-level that may soon portend harm.

VII. Related Work

In our previous survey in [1], we have carried out extensive review of of performance anomaly detection and bottleneck identification research in distributed and cloud environments. In this section, we describe some related works from the survey in addition to recent studies. Performance anomaly Detection (PAD) is concerned with identifying problem symptoms and anomalies in performance metrics of computer systems. Many recent work in this area generally focus on detecting problems in system-level metrics using predominantly statistical learning techniques. SEAD [35] adapts Support Vector Machines (SVM) to achieve a self-evolving mechanism for detecting anomalies in system metrics. EbAT [36], analyses entropy time-series of metric distributions to automatically identify anomalies in datacenter servers. Dean et al [5] propose UBL, an unsupervised framework for identifying performance anomalies in systems metrics using Self-Organizing Maps (SOMs). A workload-aware technique for localizing anomalous requests and metrics in web applications using Local Outlier Factor (LOF) and the Student-t test is presented in [6]. The limitation of these works is that they ignore the cause-effect relationship and correlation between anomalies at the service level and system-level events such as capacity bottlenecks and faults.

Moreover, existing approaches for detecting anomalies in application performance metrics (e.g. response time and throughput) either resort to simplistic mechanisms based on thresholding or require intrusive application instrumentation (such as tracing request flows in [10] and [11]). Iqbal et al in [7] [37] employ hard SLA thresholds to detect anomalies in service response times. CloudPD [8] performs automated end-to-end problem detection, diagnosis and classification of faults in cloud environments. CloudPD detects anomalous response time and throughput measurements using percentage-based thresholds. E2EProf [10] analyzes causal paths of service requests to identify end-to-end requests delays and faulty components in distributed services. Spikes in requests delays are detected by point maxima (e.g. $3\sigma+\mu$) thresholds. Zhang et al [19] detects anomaly behaviour in jobs (applications) running in a Google web-search cluster based on a 2σ deviation in the job's cycle-per-instruction (CPI) metric distribution.

The behaviour-based anomaly detection method proposed in this paper is closely related to approaches proposed in [14] and [15] for intrusion detection in network traffic. The predictive anomaly detection scheme is similar to the work of Munz et al [30] and Ye et al [38] focusing on the use of statistical control charts in anomaly detection. Time series forecasting have also been used in dynamic resource provisioning in cloud environments such as in [39] and [40].

VIII. Conclusion

We have proposed and evaluated two schemes for detecting anomalies in real-time service performance metric measurements. In use cases where deviations from a known metric baseline is classified as anomalous, we have presented a technique that automatically detects deviations from the given baseline using non-parametric statistical learning. To address cases where a baseline is unknown, we have exploited the temporal dependency in service performance metric measurements to identify anomalies. Deviations in service performance behaviour are detected by applying adaptive control limits on residuals of Holt-Winter's forecasts using EWMA control charts. Our evaluation is based on tail service latency traces obtained from experiments in a real virtualized testbed under realistic load and fault injections. Results show the applicability of our approach for anomaly detection and also suggest that many service performance anomalies may be explained by system-level events such as resource contention and capacity bottlenecks.

In the future, we plan to extend our approach for automatic root-cause analysis. By correlating system-level events with service-level anomaly detection, the resulting system may be able to diagnose observed anomalies, identify potential root-causes and remediations. Also, we plan to further evaluate accuracy of the techniques using labeled datasets from public system traces.

ACKNOWLEDGMENT

This work is supported by the Swedish Research Council (VR) under contract number C0590801 for the Cloud Control project, the Swedish Strategic Research Program eSSENCE, and the European Unions Seventh Framework Programme under grant agreement 610711 (CACTOS).

REFERENCES

[1] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance Anomaly Detection and Bottleneck Identification," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 4:1–4:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2791120

[2] T. Kelly, "Detecting Performance Anomalies in Global Applications," in *WORLDS*, vol. 5, 2005, pp. 42–47.

[3] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu, "Detecting Transient Bottlenecks in n-tier Applications through Fine-grained Analysis," in *33rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2013, pp. 31–40.

[4] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *32nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2012, pp. 285–294.

[5] D. J. Dean, H. Nguyen, and X. Gu, "UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 191–200.

[6] T. Wang, J. Wei, W. Zhang, H. Zhong, and T. Huang, "Workload-aware anomaly detection for web applications," *Journal of Systems and Software*, vol. 89, pp. 19–32, 2014.

[7] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Sla-driven Automatic Bottleneck Detection and Resolution for Read Intensive Multi-tier Applications Hosted on a Cloud," in *Advances in Grid and Pervasive Computing*. Springer, 2010, pp. 37–46.

[8] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, "Cloudpd: Problem determination and diagnosis in shared dynamic clouds," in *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–12.

[9] 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*. USENIX Association, 2014, pp. 16–16.

[10] S. Agarwala, F. Alegre, K. Schwan, and J. Mehalingham, "E2eprof: Automated end-to-end performance management for enterprise systems," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2007, pp. 749–758.

[11] R. Filipe, S. Boychenko, and F. Araujo, "On client-side bottleneck identification in http servers," in *10th International Conference on Internet and Web Applications and Services*. IARIA, 2015, pp. 22–27.

[12] P. Carden. (2013) Network design manual: Network baselining and performance management. [Online]. Available: http://www.networkcomputing.com/netdesign/base1.html

[13] N. Bhatia, "Proactive Detection of Performance problems using Adaptive thresholds," 2010. [Online]. Available: https://neerajbhatia.files.wordpress.com/2010/10/Adaptive-thresholds.pdf

[14] Y. Gu, A. McCallum, and D. Towsley, "Detecting Anomalies in Network Traffic Using Maximum Entropy Estimation," in *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*. USENIX Association, 2005, pp. 32–32.

[15] D.-Y. Yeung and C. Chow, "Parzen-window Network Intrusion Detectors," in *Proceedings of the 16th International Conference on Pattern Recognition*, vol. 4. IEEE, 2002, pp. 385–388.

[16] F. J. G. Gisbert, "Weighted Samples, Kernel Density Estimators and Convergence," *Empirical Economics*, vol. 28, no. 2, pp. 335–351, 2003.

[17] A. Moore, "A Tutorial on KD-Trees," http://www.cs.cmu.edu/simawm/papers.html, 1991, Extract from PhD Thesis.

[18] D. C. Montgomery, *Introduction to Statistical Quality Control*. John Wiley & Sons, 2007.

[19] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI2: CPU Performance Isolation for Shared Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 379–391.

[20] S. Malkowski, M. Hedwig, and C. Pu, "Experimental Evaluation of N-tier Systems: Observation and Analysis of Multi-bottlenecks," in *International Symposium on Workload Characterization, IISWC*. IEEE, 2009, pp. 118–127.

[21] C. C. Aggarwal, *Outlier Analysis*. Springer Science & Business Media, 2013.

[22] B. E. Hansen, "Lecture Notes on Nonparametrics," 2009. [Online]. Available: http://www.ssc.wisc.edu/~bhansen/718/NonParametrics1.pdf

[23] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*. CRC press, 1986, vol. 26.

[24] B. E. Hansen, "Bandwidth Selection for Nonparametric Distribution Estimation," *Manuscript, University of Wisconsin*, 2004.

[25] R. N. Riegel, "Generalized n-body problems: a framework for scalable computation," Ph.D. dissertation, Georgia Institute of Technology, 2013.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[27] R. J. Hyndman and G. Athanasopoulos, *Forecasting: principles and practice*. OTexts, 2014, accessed: 2016-02-10.

[28] P. R. Winters, "Forecasting Sales by Exponentially Weighted Moving Averages," *Management Science*, vol. 6, no. 3, pp. 324–342, 1960.

[29] R. Lawton, "How Should Additive Holt–Winters Estimates be Corrected?" *International Journal of Forecasting*, vol. 14, no. 3, pp. 393–403, 1998.

[30] G. Munz and G. Carle, "Application of Forecasting Techniques and Control Charts for Traffic Anomaly Detection," in *Proceedings of the 19th ITC Specialist Seminar on Network Usage and Traffic*. Citeseer, 2008.

[31] NIST/SEMATECH, "e-Handbook of Statistical Methods," 2012, [Online; accessed 10-February-2016]. [Online]. Available: http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc431.htm

[32] B. Welford, "Note on a Method for Calculating Corrected Sums of Squares and Products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[33] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[34] D. Desmeurs, C. Klein, A. V. Papadopoulos, and J. Tordsson, "Event-Driven Application Brownout: Reconciling High Utilization and Low Tail Response Times," in *International Conference on Cloud and Autonomic Computing (ICCAC)*, 2015.

[35] H. S. Pannu, J. Liu, and S. Fu, "A self-evolving anomaly detection framework for developing highly dependable utility clouds," in *Global Communications Conference (GLOBECOM)*. IEEE, 2012, pp. 1605–1610.

[36] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online detection of utility cloud anomalies using metric distributions," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*. IEEE, 2010, pp. 96–103.

[37] W. Iqbal, M. Dailey, and D. Carrera, "Sla-driven adaptive resource management for web applications on a heterogeneous compute cloud," in *Cloud Computing*. Springer, 2009, pp. 243–253.

[38] N. Ye, S. Vilbert, and Q. Chen, "Computer Intrusion Detection through EWMA for Autocorrelated and Uncorrelated Data," *IEEE Transactions on Reliability*, vol. 52, no. 1, pp. 75–82, 2003.

[39] H. Engelbrecht and M. van Greunen, "Forecasting Methods for Cloud Hosted Resources, a Comparison," in *11th International Conference on Network and Service Management (CNSM)*. IEEE, 2015, pp. 29–35.

[40] C. Vazquez, R. Krishnan, and E. John, "Time Series Forecasting of Cloud Data Center Workloads for Dynamic Resource Provisioning," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, vol. 6, no. 3, pp. 87–110, 2015.