EVALUATING CLOUD AUTO-SCALER RESOURCE ALLOCATION PLANNING UNDER MULTIPLE HIGH-PERFORMANCE COMPUTING SCENARIOS

by

KESTER LEOCHICO, B.S.

THESIS
Presented to the Graduate Faculty of
The University of Texas at San Antonio
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

COMMITTEE MEMBERS:
Eugene John, Ph.D., Chair
Wei-Ming Lin, Ph.D.
Wonjun Lee, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Engineering
Department of Electrical and Computer Engineering
August 2017

DEDICATION

To all of my friends and family who have been there with me the entire time. This is for you.

ACKNOWLEDGMENTS

The author would like to thank Ahmed Ali-Eldin Hassan from Umeå University for providing the original source code for the Python-based event simulator used in his research, source code for the auto-scaling algorithms used in this thesis, and the Wikipedia web traces that he used for his research. This thesis would not be possible without his support and advice.

August 2017

EVALUATING CLOUD AUTO-SCALER RESOURCE ALLOCATION PLANNING UNDER MULTIPLE HIGH-PERFORMANCE COMPUTING SCENARIOS

Kester Leochico, M.S.
The University of Texas at San Antonio, 2017

Supervising Professor: Eugene John, Ph.D.

Cloud computing enables users to elastically acquire as many computing resources as they need on-demand while minimizing resource allocation to what is needed on the provider side, allowing users to acquire only the resources that they need while reducing the costs of acquiring said resources over traditional datacenters. The key to enabling these improvements are cloud auto-scalers, which are subsystems that are responsible for planning out how many resources to provision in response to current/future demand. The current state of the art in terms of comparing different auto-scaling algorithms is immature, however, due to the lack of consistent, standard evaluation methodologies that analyze cloud auto-scalers under multiple scenarios, making it difficult to compare the results between proposed auto-scalers. In an effort to address some of these issues, this work analyzes the effects of changing the workload mix and lowering the average job runtime (as represented by the service rate) on the performance of three cloud auto-scalers from the literature in an effort to better model the behavior of cloud auto-scalers under high-performance computing scenarios involving long-running jobs rather than short-lived jobs as in previous studies. The simulation and analysis was carried out using the Performance Evaluation framework for Auto-Scaling (PEAS), a cloud simulator framework for modelling the performance of cloud auto-scaling algorithms using scenario theory.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE: INTRODUCTION

Substantial parts of this thesis are based on the work described in [1] *and* [2]. *Chapter 1 is based on the introduction of* [2]. *Chapter 2 is original with the exception of the definition of elasticity, which is based on information from the introduction of* [2]. *Chapter 3 is original with the exception of Section 3.4, which is based on the introduction of* [1]*, and Section 3.5, which is based on the background information sections of* [1] *and* [2]. *Chapter 4 is based on information that was previously presented in* [1] *and* [2]*, with more information given on specific workloads used. Section 5.2 of Chapter 5 contains research data that was previously presented in* [2]*, although Section 5.1 is original. Chapter 6 is substantially original.*

Cloud computing, a high performance computing paradigm in which dynamically scalable and virtualized computing resources are provisioned to remote customers over the Internet as a service on an as-needed basis [3] [4] [5], represents a major shift in the way customers and businesses pay for computing resources. By pairing the older software-as-a-service (SaaS) notion of client-server computing with the idea of utility computing (the notion of paying for access to computing resources on a pay-as-you-go, as-needed basis), it gives customers access to virtually unlimited amounts of computing resources without the overhead of running and maintaining their own servers, allowing companies to start small and scale up readily as their computing requirements increase [6].

The key property of a cloud computing system that enables the utility computing paradigm that is central to cloud computing is *elasticity*. Not to be confused with the notion of *scalability* (which merely deals with the ability of a system to scale up to meet large workload requirements, regardless of how responsively it does so), elasticity is the property of a computing system or application to dynamically and automatically allocate computing resources at runtime in a timely, responsive manner based on the current resource demand [3] [7] [8] [9]. This quality is a large part of what enables clouds to provide the economic benefits that they do; by scaling quickly, one can respond to changes in the load without violating service-level agreements (SLAs), and by provisioning only as many resources as the system needs to satisfy SLAs, one can reduce the cost (both economic and environmental) of acquiring and using computing resources [6].

Because this is a very difficult task to handle properly when done manually, the task of managing elasticity is typically assigned to cloud auto-scalers, which are subsystems within a cloud that decide how many computing resources to provision in response to current/future demand [9]. However, the current state of the art with respect to auto-scaler evaluation is immature and inconsistent, due to a lack of consistent benchmarking metrics [3]; a lack of formal, standard evaluation methodologies for auto-scaling algorithms [3]; and the use of only a few short workloads to characterize auto-scaler behavior in much of the literature [10]. As a result, the overall lack of consistency between performance evaluation methods makes it difficult to compare the performance of any two different auto-scalers, because the research methodologies can vary wildly between papers and thus makes it difficult to determine whether the experimental results observed by the authors will actually hold up to real-world usage.

One proposed solution to the issues facing cloud auto-scaling evaluation is the Performance Evaluation framework for Auto-Scaling (PEAS), first proposed in [10], which aims to address the aforementioned issues with the current state-of-the-art in cloud auto-scaler performance evaluation by providing a formalized, standardized, and mathematically rigorous methodology for providing probabilistic guarantees on the goodness of an auto-scaler's performance based on its distance from the ideal capacity outlay. PEAS achieves these aims in part by running as many as hundreds of workload traces to account for the effects of the workload profile on auto-scaler behavior.

In this thesis, the effects of changing the workload mix and the service rate used in the experiment (which affects how quickly/slowly jobs are processed in the simulation) are explored. The goal is to consider how the auto-scaling algorithms described in [10] perform in terms of their ability to accurately match computing resources (in the form of virtual machines) to current demand (in the form of incoming requests) under alternate scenarios. Specifically, [10] only considered the performance of cloud auto-scaling algorithms when running under conditions that modeled a web server deployment. However, the results that the PEAS framework provides only hold if the conditions experienced in real world usage are similar to the workload mixes used in simulation. In practice, cloud computing systems can run a wide variety of workload sets, such as scientific computing workloads and data warehousing [11], and this thesis aims to broaden the original experiments by considering the effects of using alternate workload sources as well as lowering the service rate of the virtual machines (VMs) used in the simulation to more closely model the job runtimes used in the aforementioned traces.

The rest of this thesis is organized as follows: Chapter 2 provides the requisite background information on the cloud computing paradigm, elasticity, and cloud auto-scalers. Chapter 3 covers background information on cloud auto-scaler performance evaluation schemes in the literature and an overview of the PEAS framework. Chapter 4 describes the research methodology used for the experiment, including the simulation parameters, workloads, and tools involved. Chapter 5 covers the results as well as commentary on them. Finally, Chapter 6 provides concluding remarks on the study as well as proposals for further research.

CHAPTER TWO: BACKGROUND INFORMATION

2.1: Cloud Computing

Cloud computing is a computing paradigm in which dynamically scalable and virtualized computing resources are rapidly and automatically provisioned and released to remote customers over the Internet as a service on an as-needed, pay-as-you-go basis, so called as a reference to the cloud drawing that has been historically used to represent telephone networks and Internet infrastructure in diagrams [5] [4] [3] [12]. It refers to both application software provided as services over the Internet as well as the hardware and system software that powers the data centers providing these services [6]. Cloud computing is defined by the following essential characteristics [12]:

- *On-demand self-service:* Computing resources are automatically provisioned and controlled by consumers with no human interaction from the service provider required.
- *Broad network access:* Computing resources are made available via computer networks to any number of heterogeneous computing devices via standard protocols
- *Resource pooling:* Computing resources (such as storage, processing, memory, and network bandwidth) are pooled together between multiple customers via a multi-tenant model in which physical and virtual resources are dynamically (re-)assigned based on current demand. Each customer does not know the exact location of the resources allocated to them but can specify the general vicinity of the allocated resources.
- *Rapid elasticity:* Computing resources are dynamically and automatically allocated to scale with current demand, providing customers with the illusion of unlimited on-demand computing resources.

- *Measured service:* Computing resource usage is monitored, optimized, and reported transparently via a metering capability (typically pay-per-use or charge-per-use) for the benefit of both cloud providers and customers.

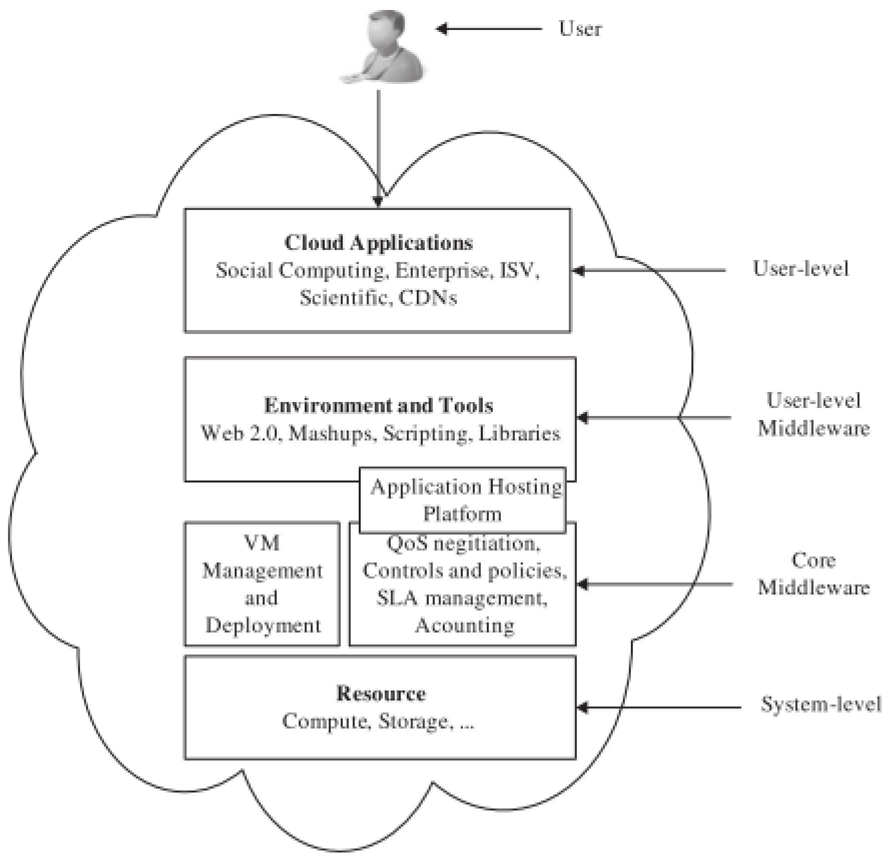
User
Cloud Applications
Social Computing, Enterprise, ISV, Scientific, CDNs
User-level
Environment and Tools
Web 2.0, Mashups, Scripting, Libraries
User-level Middleware
Application Hosting Platform
VM Management and Deployment
QoS negitiation, Controls and policies, SLA management, Acounting
Core Middleware
Resource
Compute, Storage, ...
System-level

Figure 2.1: Cloud Computing [4]

Cloud computing can be seen as the product of years of evolution in the development of distributed computing paradigm, in which large amounts of computing nodes are connected together via computer networks to achieve reliability, scalability, and information sharing and exchange [4]. Previous distributed computing paradigms include:

- *Cluster*: Clusters are HPC systems in which commodity computers, storage devices, and networking hardware are joined together to form a single high-availability system that provides the illusion of a single computer resource. They are used for compute-intensive applications and in replicated storage/backup servers that provide fault-tolerant and reliable storage for critical parallel applications [4] [13].

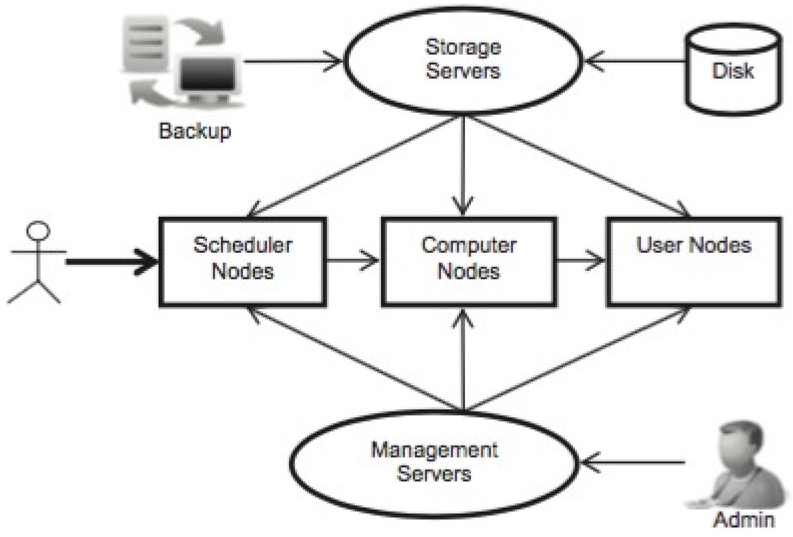

Figure 2.2: Cluster Computing [4]

- *Grid*: Grids are HPC systems in which local clusters are connected through the Internet to work with non-interactive workloads involving large amounts of files in loosely coupled, heterogeneous, and geographically dispersed computing systems. They enable virtual organizations (groups of individuals/organizations that agree upon resource sharing rules) to share resources and solve problems. Much like cloud computing, it enables on-demand resource provisioning with pay-as-you-go utility pricing for users and can therefore be seen as a precursor to the cloud computing paradigm [4] [13] [14].

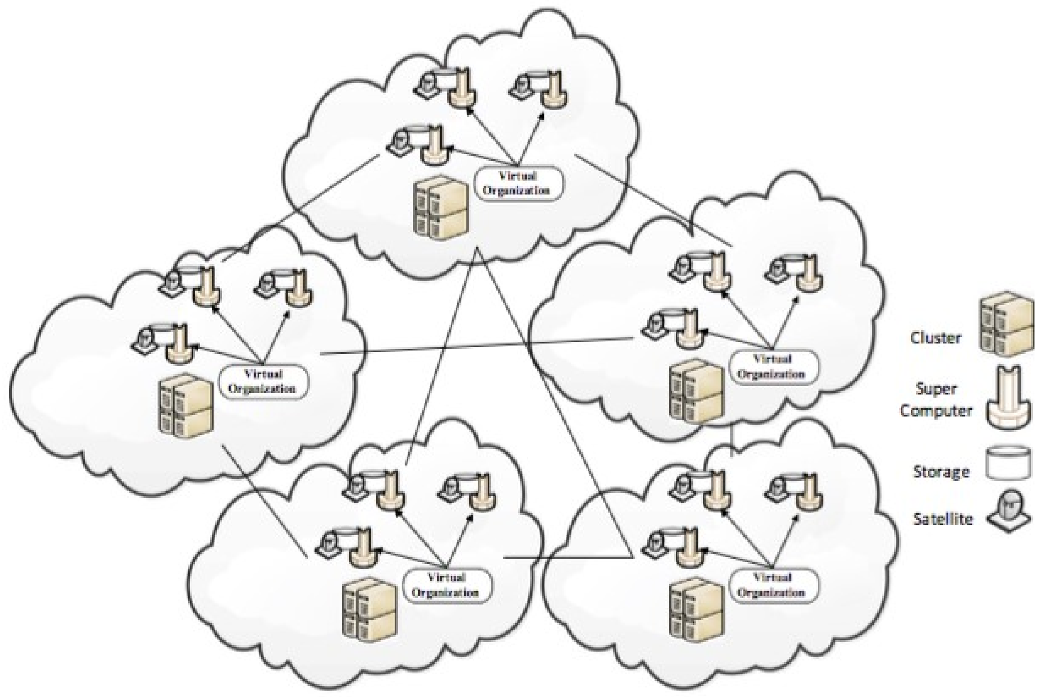

Figure 2.3: Grid Computing [4]

In cloud computing, a *provider* sells access to virtually unlimited amounts of computing resources, whether in the form of virtual machines or web-hosted applications/APIs, to *clients* hosting applications on the cloud or *users* accessing said applications. Cloud providers charge end-users based on the types of VMs allocated, unit charge, and so forth while agreeing with clients and users to maintain a certain level of quality of service (QoS). Such agreements take place in the form of a service level agreement (SLA), which is a contract between either cloud providers and clients (in the case of resource SLAs) or clients and end users (in the case of application SLAs) to guarantee a certain level of performance. The level of performance is characterized in terms of metrics such as the response time or the infrastructure availability [9].

Cloud computing services are provided at different levels of abstraction from the bare metal depending on the cloud provider. Different cloud computing services provide different tradeoffs between flexibility/portability (via greater access to lower-level, closer-to-the-metal

APIs) and the amount of available built-in functionality provided by the cloud provider (via access to higher-level, more heavily abstracted APIs) [5]. This is best illustrated by referring to the definitions of cloud service models as defined by the National Institute of Standards and Technology (NIST) [12]:

- *Infrastructure as a Service (IaaS):* The IaaS model provides the lowest-level access to computing resources under the NIST model. IaaS clouds provide access to computing and networking resources along with the management software to handle them by instantiating virtual machines of pre-determined computing power and memory size, OS, and installed libraries on-demand [9]. This gives users low-level control over the OS, VM storage, and deployed applications along with networking components like the firewall. Examples include Amazon Elastic Compute Cloud (Amazon EC2), Eucalyptus, and E-learning Ecosystem.
- *Platform as a Service (PaaS):* Instead of exposing low-level access to computing resource via VMs as in the IaaS model, the PaaS model instead provides access to cloud infrastructure via a solution stack that provides the programming environments (programming languages, libraries, services, etc.) and tools for deploying applications on a given cloud provider's infrastructure [13]. Users have no control over the underlying infrastructure, but they do have control over the cloud-hosted application and possibly the configuration settings for the underlying application-hosting environment. Examples include Microsoft Azure, Amazon Elastic Beanstalk, and Google App Engine.
- *Software as a Service (SaaS):* While the PaaS model provides access to developer tools for creating cloud applications, the SaaS model provides finished cloud applications to users over a computer network. Such cloud applications are made available either

through a thin client interface such as a web browser (i.e. webmail) or through APIs, allowing for on-demand access to applications [15]. Because cloud access is exposed through end-user applications under this model rather than developer middleware, users do not have any control over any of the underlying infrastructure and can only control the applications via provider-exposed settings.

While the software as a service notion of pairing powerful server hardware with relatively weak thin clients is not new, as is the whole notion of distributed computing, cloud computing was only made possible in recent years by the emergence of commoditized computer hardware and software (especially fast virtualization and standardized software stacks) as well as ubiquitous broadband Internet access. These developments, along with the growth in demand for computing resources, led to the emergence of massive datacenters built out of large amounts of commodity computing hardware. This, in turn, enabled enormous improvements in economy of scale: it is 5-7× cheaper (in terms of electricity, bandwidth, operations, software, and hardware costs) to build a massive datacenter (tens of thousands of computers) than it is to build a medium-sized datacenter (hundreds of thousands of computers). These improvements in economies of scale, along with statistical multiplexing to improve utilization over traditional datacenters, enable cloud providers to offer services at less than the costs of a medium-sized datacenter while maintaining good profit margins [6] [5].

Cloud computing leverages the improvements in cost from economies of scale to better enable *utility computing*, a computing paradigm (previously seen in grid computing) which users pay for access to virtually unlimited amounts of computing resources (such as hardware infrastructure, development platforms, and applications) on an as-needed, pay-as-you-go basis. In utility computing, computing resources are paid for on a short-term, as-needed basis—users need only pay for what they need, when they need it. Resources that are not needed are released, rewarding conservation of resources [6] [3]. This key property enables many of the economic benefits of cloud computing:

- *Time-variant service demand:* Because resources are allocated as-needed, resources can be made available to meet current demand instead of allocating for peak demand (which leads to underutilization most of the time), leading to potential savings even if the hourly rate for renting a virtual machine is more than the rate to own a computer [6].
- *Unknown advance demand:* Because resources are allocated as-needed, this eliminates the need for an up-front commitment to a large initial investment by users; this allows users to start small and scale up rapidly as their resource requirements increase [6] [5].
- *Converting capital expenditures to operational expenditures (CapEx to OpEx):* Because cloud computing eliminates the need for large initial up-front investments in capital (such as servers, networking hardware, etc.), this allows money that would be spent on capital to instead be reinvested elsewhere. In addition, users can take advantage of the reductions in up-front capital costs to deploy hybrid computing use cases in which private clouds (datacenters operating under the same protocols as a public cloud) work in tandem with public cloud providers to offload compute tasks to the public cloud during surge use cases [5] [6].
- *Cost associativity:* Cloud computing enables users to finish long-running batch analysis tasks faster by enabling them to rent multiple VMs for fewer hours at the same cost as renting one computer for many hours.

The strong economic case for cloud computing—pay-as-you-go access to elastic, scalable computing resources at a low up-front cost—enabled by utility computing and the cloud computing paradigm allows customers to run high-performance computing (HPC) use-cases that were traditionally the domain of on-premises datacenters (such as the ones listed in Table 2.1) onto remote cloud computing deployments instead.

Table 2.1: Common workload categories [11]

Web Serving	Static and dynamic web content serving, streaming media, RSS, mash-ups and SMS
Web Applications	Web service-enabled applications, eCommerce, eBusiness, Java application servers, Rich Internet Applications (Adobe Flash, JavaFX, MS Silverlight) and web search engine applications
Business Intelligence And Data Warehouse	Data mining, warehousing, streaming data analytics, text mining, competitive analysis, business intelligence and business decision applications
ERP And CRM	Enterprise resource planning (ERP) and scheduling, engineering and manufacturing planning and scheduling, supply chain management applications, purchase order management, finance applications, customer relationship management (CRM) and HR applications
Analytics	Online analytic processing (OLAP), business optimization, marketing and sales forecasting, management reporting, risk management and analysis applications, credit scoring and portfolio analysis
Numerical And Batch	Engineering design and analysis, scientific applications, high performance computing, Monte Carlo-type simulations, medical image processing and floating-point intensive batch computations
Collaboration	Web 2.0 applications for online sharing and collaboration, instant messaging (IM), mail servers (SMTP) and Voice over Internet Protocol (VoIP)
File And Print	Print, file systems, archival and retrieval
Desktop	Desktop-based computing, desktop service and support applications, and desktop management applications
Development And Test	Development and test processes and image management

2.2: Elasticity and Cloud Computing

Perhaps more than any other property of cloud computing, the concept of *elasticity* is the one that's most fundamental to the cloud computing paradigm. Not to be confused with *scalability* (the ability of a computing system to scale up to large workload requirements independent of the time with which it does so), elasticity is the property of a computing system that allows resources to be dynamically and automatically allocated at runtime in a timely, responsive manner based on current workload demand, especially during periods of peak usage. Whereas scalability deals only with the ability to scale up, elasticity deals with both the ability to scale up *and* down responsively; the faster the better. While an elastic computing system is also implied to be a scalable one below a given upper bound, a manually scalable computing system is not, by definition, elastic due to the need for the scaling to be automatic [9] [3] [8].

Elasticity is a key aspect of maintaining quality of service on the cloud [3]. Instead of waiting to add extra computers to scale up to current demand, resources can be added or removed in real time (minutes rather than weeks) and in a fine-grained manner. This allows resources to be matched more closely to the current workload at runtime and without interruption of service [6]. This also provides the illusion of unlimited on-demand computing resources that is the key to the utility computing paradigm [3].

Elasticity also enables users to reduce underutilization of clouds over conventional data centers. Ideally, resource allocation should be closely matched to the current requirements such that quality-of-service requirements are maintained at the lowest possible cost [10]. However, because conventional data centers must over-provision for peak workload (which can exceed the average workload by factors of 2-10 for most services), they typically have average utilization rates at 5-20% of capacity. In addition, the difficulty of correctly predicting seasonal/periodic/unexpected spikes in usage and correctly provisioning computing resources in advance can result in either under-provisioning or over-provisioning of services, both of which are costly in terms of resource cost and user satisfaction respectively [6].

Elasticity can be achieved through any combination of *horizontal scaling, vertical scaling*, and *migration*. Horizontal scaling (also known as replication) involves instantiating new instances (virtual machines, applications, containers, etc.) to meet current demand. It is commonly used in production cloud environments, where it is the most commonly used scheme for providing elasticity. However, it can also result in extra overhead from unnecessary over-provisioning due to the fact that the unit of scaling is a whole new instance, and each instantiation of such an instance has its own resource requirements [9] [3].

Vertical scaling, by contrast, involves adding/removing assigned computing resources (CPUs, memory, storage, etc.) to an existing instance; this is less common due to the lack of operating system support for on-the-fly CPU/memory/storage reallocation without restarting the virtual machine instance [9]. Vertical scaling can be implemented via either *resizing* instances or *replacing* instances. Resizing instances involves changing the amount of assigned resources for a given instance at runtime. This approach is common in UNIX-like operating systems that can support changing the amount of assigned CPUs/memory without restarting (i.e. via memory ballooning). Replacing instances, by contrast, involves replacing a pre-existing instance (typically a virtual machine) with a larger instance. This is the more common approach among commercially available IaaS providers, including Amazon and GoGrid [3].

The third approach to achieving elasticity in clouds is via *migration*, in which instances are transferred from one physical server to another for consolidation and load-balancing purposes. This approach tends to be focused more on balancing the load between physical machines rather than maintaining performance of individual applications or virtual machines [3].

2.3: Cloud Auto-Scalers and the MAPE Loop

Because of the difficulty of manually provisioning resources correctly, the task of achieving elasticity should ideally be handled by an auto-scaling system, which are cloud computing subsystems that determine the amount of computing resources to provision to users in response to current and/or future demand [9]. In general, cloud auto-scalers seek to balance satisfying the application SLA against the cost of allocating cloud resources. In particular, they must find a way to avoid *under-provisioning*, *over-provisioning*, and *oscillation*.

Under-provisioning occurs when the cloud computing system does not have enough allocated computing resources to resolve all incoming demands under the current SLA. This can occur because processing requests for additional computing resources takes time before the resources are made available. In particular, traffic bursts under an already under-provisioned cloud system can result in SLA violations and system congestion, and resolving the issue can take time [9].

Over-provisioning occurs when the cloud computing system has more resources than are needed to resolve all incoming demands under the current SLA. While a certain degree of over-provisioning is desirable to cope with small workload variations, it is something that should be minimized where possible due to the unnecessary cost of provisioning those extra resources for the end-user [9].

Oscillation occurs when the cloud computing system carries out scaling actions too quickly before the effects of each scaling action can be observed. This is typically kept under control by using a capacity buffer (keeping instances somewhat underutilized instead of running at capacity) or by using a cooldown period [9].

In addition, the ideal cloud auto-scaler should be *scalable*, *adaptive*, *responsive*, *robust*, and have *awareness of QoS and cost considerations* [10]:

- *Scalability:* Scalability deals with the ability of a computing system or application to scale up to meet large workload requirements. For a computing system to be scalable, it must have both *platform scalability* (the computing system must be scalable) and *application scalability* (the application can meet the performance goals specified by the service-level agreement [SLA] even in the face of workload intensity increases). A scalable system that satisfies both the platform and application scalability requirements lacks both active and passive resource bottlenecks, but is in practice limited by either workload intensity (workloads cannot be too large) or resource pool size (there has to be enough computing resources available to meet the requirements of the current workload [8].

- *Adaptiveness:* The ideal cloud auto-scaler should be able to adapt to the ever-changing dynamics of internet and cloud workloads.
- *Responsiveness:* Auto-scaling algorithms need to be able to determine the required capacity quickly enough to preserve QoS requirements in the face of dynamic workloads.
- *Robustness:* The ideal cloud auto-scaler should be stable with respect to changing workloads and system dynamics and should resist oscillation.
- *QoS and Cost Awareness:* Auto-scaling algorithms should be aware of QoS concerns and balancing the costs of over-provisioning vs. the cost of under-provisioning.

Cloud auto-scaling processes follow the MAPE loop that has been observed in other autonomous systems, which consists of **M**onitoring, **A**nalysis, **P**lanning, and **E**xecution phases. In the MAPE loop, cloud auto-scalers *monitor* resource utilization data to ensure SLA compliance, which is then used to *analyze* the data and *plan* whether to scale up, scale down, or not scale. The cloud auto-scaler then *executes* the planned scaling operations. A more detailed description of each phase is as follows [9]:

- *Monitoring:* In the monitoring phase, the cloud auto-scaler collects data on computing resource utilization metrics (such as those listed in Table 2.2) that are provided by a monitoring system in order to ensure compliance with the SLA. These performance metrics are used to provide the auto-scaler with correct data on current utilization. While auto-scaling algorithms deal mostly with the analysis and planning phases of the MAPE loop, the performance of the auto-scaler does depend on the quality of the metrics, the sampling granularity, and the overhead of obtaining the metrics.

Table 2.2: List of cloud monitoring metrics [9]

Hardware	CPU utilization per VM, disk access, network interface access, memory usage.
General OS Process	CPU-time per process, page faults, real memory (resident set).
Load Balancer	size of request queue length, session rate, number of current sessions, transmitted bytes, number of denied requests, number of errors.
Application Server	total thread count, active thread count, used memory, session count, processed requests, pending requests, dropped requests, response time.
Database	number of active threads, number of transactions in a particular state (write, commit, roll-back).
Message Queue	average number of jobs in the queue, job's queuing time.

- *Analysis:* In the analysis phase, the cloud auto-scaler analyzes the utilization data from the monitoring system and uses the data to determine the current system utilization as well as predict the future system utilization if the auto-scaling algorithm calls for it. The analysis can be *reactive* (analyzing based on current demand only) and/or *proactive* (analyzing based on predicted future demand). Proactive utilization analysis is important because of the delay between when an auto-scaling action is started and when it takes effect (i.e. VM startup time). In the event of sudden traffic bursts, a purely reactive system may not be able to scale responsively enough to cope with the demand due to this delay, so being able to predict future demand in advance can be helpful in order to anticipate upticks in demand.

- *Planning:* The planning phase involves deciding on how many resources (memory, number of VMs, etc.) to assign to/remove from an application in order to resolve both cost and SLA compliance issues. Planning decisions are made based on data generated during the monitoring and analysis phases, the target SLA, and cloud infrastructure considerations.
- *Execution:* The planned scaling actions are carried out in the execution phase. The actual implementation of the scaling action is carried out through the cloud provider's API and is abstracted away from end-users. Note that the actual resource provisioning takes time, and that delays on resource provisioning can be part of the resource SLA.

[9] categorizes cloud auto-scalers into five general classes of algorithms based on the underlying theory behind their design and operation:

- *Threshold-based Rules:* Threshold-based auto-scalers allocate resources based on whether a certain metric has exceeded or gone below a set of thresholds or not. While it is a popular choice for commercial cloud providers due to its simple approach, the thresholds are application-specific and must be attuned to the specific workload nature and trends.
- *Reinforcement Learning:* Reinforcement learning-based auto-scalers decide scaling decisions given a current state (input workload, performance, etc.) based on a reward mechanism. The level of rewards given per state and action are determined based on previous behavior via a value function $Q(s, a)$ known as the Q-value function. The policy determined by the auto-scaler is the one that maximizes the Q-value for each state.
- *Queuing Theory:* Queuing theory-based auto-scalers apply mathematical queuing models (typically used to analyze systems characterized by static arrival and service rates) to

derive performance metrics based on the queuing model and the arrival/service/response times.

- *Control Theory:* Control theory-based auto-scalers apply control theory to design control systems that automatically scale the system. These can include open-loop (no feedback), feedback (observes system output and corrects deviation from the desired output), or feed-forward (using models of the system behavior to anticipate errors in the output and react before the error occurs) control systems. Fixed-gain (i.e., PID controllers et al.), adaptive (self-tuning, gain-scheduling, etc.) or model predictive systems have also been proposed.

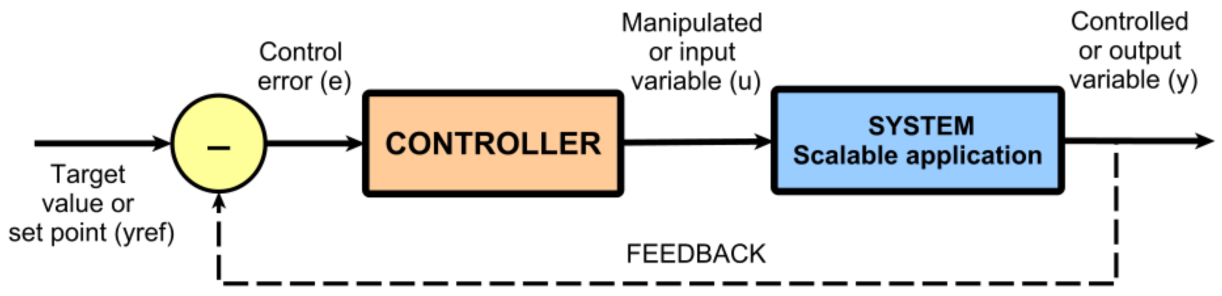
Control error (e)
Manipulated or input variable (u)
Controlled or output variable (y)
–
CONTROLLER
SYSTEM Scalable application
Target value or set point (yref)
FEEDBACK

Figure 2.4: Block diagram of a feedback control system [9]

- *Time Series Analysis:* Time series analysis-based auto-scalers involve the use of prediction techniques based on analyzing *time series* (sequences of data points sampled at successive points in time) to forecast future workloads/resource usage and use those predictions to plan an auto-scaling action accordingly. These auto-scalers are promising due to their ability to predict future demand, and can be used in conjunction with reactive auto-scaling techniques to proactively reconfigure the system to avoid resource shortages. However, they suffer from prediction accuracy that is sensitive to the target application, the nature of the input workload, the metric used, the history window and prediction interval, and the specific technique used.

CHAPTER THREE: CLOUD AUTO-SCALER PERFORMANCE EVALUATION

Cloud auto-scalers in the literature can be, and have been, evaluated under a wide variety of different experimental configurations that can make it difficult to compare the performance of different auto-scalers based only on the results given in any given paper. A variety of cloud benchmarking schemes, workloads, and experimental platforms have been used in the literature to evaluate cloud auto-scalers.

3.1: Experimental Platforms

Cloud auto-scalers are evaluated on an *experimental platform* that in some way models or mirrors the behavior of a production cloud platform. An experimental platform can either be an actual *production* cloud, a *custom testbed*, or a *simulator*.

Production platforms are real cloud deployments, either from a commercial cloud provider or a private cloud deployment. They allow proposed cloud auto-scalers to be tested under real-world conditions to prove their applicability under actual scenarios, but the production nature of such deployments makes them costly and time-consuming for rapid iteration of auto-scaler designs. Production experimental platforms require the experimental scenario to be reset every time a new experiment is run. In addition, it requires the entire experimental configuration to be set up if the experiment is being conducted under a public cloud provider, and it requires paying the cloud provider every time a new experiment is conducted. In addition, external factors (application interference, VM consolidation processes, etc.) cannot be controlled for in a production environment and can therefore distort the results. Finally, because the experiments run in real time, the time required for finishing experiments is longer than it would be in a simulator [9].

Custom testbeds, by contrast, are private mockups of cloud deployments that include the server hardware, virtualization software, and cloud middleware. The virtualization software used is a server-level hypervisor/virtual machine monitor (VMM) such as Xen, VMWare ESXi, or KVM, while the cloud middleware is typically provided as part of ready-made software packages for deploying private clouds. These include OpenStack (an open-source initiative supported by cloud-related companies), Eucalyptus (an open-source software package for building IaaS clouds that emulates the Amazon EC2 SOAP and query interface) [13], and Vcloud Director (commercial cloud middleware developed by VMware). While custom testbeds have lower experimentation costs (in terms of time, monetary cost, etc.) and provide better control over the testing environment than a production platform, they require spending money on server hardware and procuring/configuring cloud middleware [9].

Simulators are software tools for simulating the operation of a cloud platform, including resource allocation/deallocation, VM execution, monitoring, and other cloud management tasks. They can be either self-developed or adapted from a pre-existing simulator. Due to their abstracted, software-based nature, simulators have faster runtimes, greater control over the configuration of the experiment, and greater access to system state or performance metrics. This allows rapid iteration of auto-scaling algorithms, the testing of multiple algorithms without having to reconfigure the infrastructure, and the ability to conduct experiments without having to account for uncontrollable external factors. However, simulators require an initial investment to develop the software, and because they can only provide an abstracted model of actual cloud deployments, the reliability of the results is sensitive to the level of implementation detail provided by the simulator [9].

3.2: Cloud Benchmarking

Benchmarking is the process of evaluating the performance and other (non-functional) characteristics of computing systems for the purposes of comparing them with other systems or with industry standards. It is used to make informed purchases of computing systems through verifiable results provided by system vendors and third parties; as an aid in system design, tuning, and operation; and as a tool for training new users [16].

Benchmarking is carried out by way of *benchmarks*, which are programs that are used to provide performance measurements of systems [17]. They can be categorized into application benchmarks, synthetic benchmarks, and microbenchmarks. *Application benchmarks* consist of real-world cloud software applications coupled with a workload generator that generates requests for the app. They are used as representative workloads of real commercial cloud systems, and can be run on top of either a public cloud or a custom testbed in order to evaluate the system performance [9]. *Synthetic benchmarks*, by contrast, use representative workloads and operations to simulate typical cloud applications [3]. Finally, *microbenchmarks* consist of simple benchmarking programs that target a specific component or basic feature of clouds [3].

3.3: Workloads

Workloads are traces of the physical system utilization over time in terms of *requests* (an individual usage of a software service that is submitted by a user) sent by one or more *request classes* (categories of requests characterized by statistically indistinguishable resource demands) [8] [9]. They are represented in terms of *time series*, which are discrete functions representing sets of real-valued measurements for every time point in a finite set of equidistant time points. In the case of workloads, these time series represent a set of the number of unique arrivals during each time point, forming a *time series of request arrival rates* [8].

Workload traces are typically used in cloud computing research as inputs to simulators such as CloudSim [18] or benchmarking tools such as WikiBench and generally aim to model the behavior of real-world representative workloads. Cloud workloads can be categorized into *batch* workloads consisting of long-running, non-interactive applications (scientific computing, video transcoding, etc.) and *transactional* workloads such as web applications and web serving operations [9].

Cloud workloads can either be obtained from traces from real cloud/HPC applications or generated synthetically using a synthetic workload generator, a program that is used to generate workload traces, often representative in some way of real-world use cases, for running via a simulator or benchmarking tool. A major challenge in conducting cloud computing research is the dearth of 'proper' production cloud traces that are publicly available for use by researchers. This is due to the fact that no cloud provider (with the exception of the Google cluster traces) has provided publicly available cloud workloads [10] [9]. Due to the dearth of 'proper' production cloud traces, this often means using workload traces from representative workload categories such as the ones listed in Table 3.1.

Table 3.1: Characteristic computing resources for workload categories for the cloud [19]

Workload Category	**User View or Example Providers**	**Limiting Resources**	**Level of Cloud Relevance**
Big streaming data	Netflix	Network bandwidth	Heavy
Big database creation and calculation	Google, US Census	Persistent storage, computational capability, caching	Heavy
Big database search and access	US Census, Google, online shopping, online reservations	Persistent storage, network, caching	Heavy

Table 3.1: Continued

Big data storage	Rackspace, Softlayer, Livedrive, Zip Cloud, Sugarsync, MyPC	Persistent storage, caching, bus speed	Heavy
In-memory database	Redis, SAP HANA, Oracle In-Memory DB	Main memory size, caching	Heavy
Many tiny tasks (ants)	Simple games, word or phrase translators, dictionary	Network, many processors	Heavy
Tightly coupled calculation-intensive HPC	Large numerical modelling	Processor speed, processor-to-processor communication	Medium
Separable calculation-intensive HPC	CCI on Amazon Web Services, Cyclone (SGI) Large Simulations	Processor assignment and computational capability	Heavy
Highly interactive single person	Terminal access, server administration, web browsing, single-player online gaming	Network (latency)	Some
Highly interactive multi-person jobs	Collaborative online environment, e.g., Google Docs, Facebook, online forums, online multiplayer gaming	Network (latency), processor assignments (for VMs)	Medium
Single computer intensive jobs	EDA tools (logic simulation, circuit simulation, board layout)	Computational capability	None
Private local tasks	Offline tasks	Persistent storage	None
Slow communication	E-mail, blog	Network, cache (swapping jobs)	Some
Real-time local tasks	Any home security system	Network	None
Location aware computing	Travel guidance	Local input hardware ports	Varies
Real-time geographically dispersed	Remote machinery or vehicle control	Network	Light now, but may change in the future
Access Control	PayPal	Network	Some, light

Table 3.1: Continued

Voice or video over IP	Skype, SIP, Google Hangout	Network	Varies

Alternatively, synthetic workload generators can be used to generate synthetic workloads in lieu of real workload traces for experimental purposes. While good for controlled experimentation, the workloads generated by such software may not be realistic enough to model real world use cases [9]. A list of workload generators is provided in Table 3.2.

Table 3.2: List of workload generators

Fabian	Markov chain-based workload generator; included as part of the CloudStone benchmark stack [9]
Apache JMeter	Workload generator implemented in Java and used for load testing and performance measuring. Used to test performance on both static and dynamic resources, as well as generate heavy loads for a server, network, or object for stress-testing or overall performance evaluation purposes [9]
Rain	Workload generator toolkit using parameterizable statistical distributions to model different workload classes [9]
Httperf	Tool for generating HTTP workloads and measuring web server performance [9]
ATLAS	Workload generator that uses a data model derived from the workload behavior of high-performance scientific computing jobs that were run on the Nordic Data Grid Facility (NDGF) as a part of the ATLAS project at the European Organization for Nuclear Research (CERN) [20].

Cloud computing environments can exhibit one of four types of workload patterns (Figure 3.1). *Stable* workloads exhibit a constant number of requests per unit of time. *Growing* workloads exhibit rapidly increasing load, such as when there is a increase in web traffic for content that suddenly becomes popular. A *cyclic/bursting* traffic pattern exhibits periodic behavior with occasional bursts of traffic, such as cyclical workloads like online shopping. Finally, *on-and-off* traffic patterns fluctuate between relatively short periods of heavy activity and longer periods of light activity, such as that exhibited by batch processing workloads [21] [9].

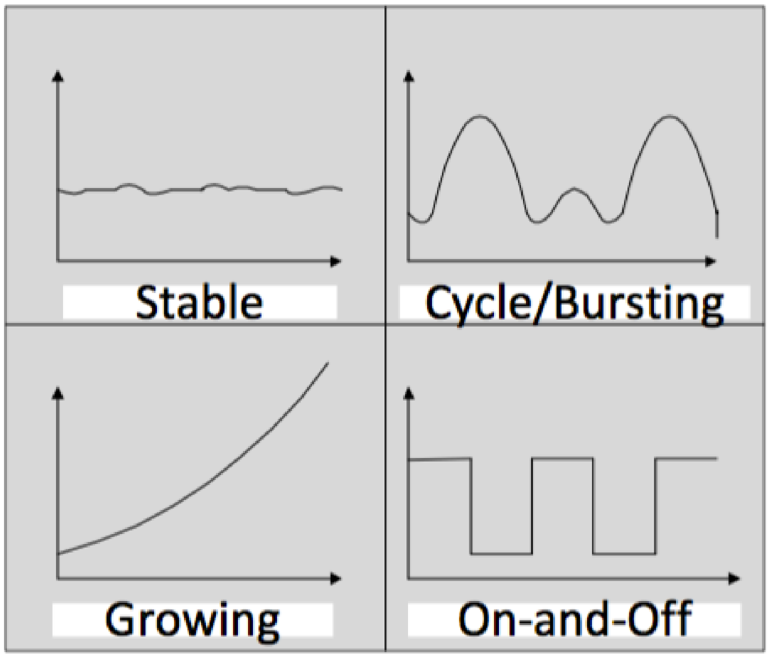

Figure 3.1: Types of workload patterns in cloud environments [21]

Of particular importance to cloud auto-scaler performance is the presence of *workload bursts*. Bursts, also known as *spikes* or *flash crowds*, are sudden increases in demand on object(s) hosted on online servers due to an increase in the number of requests or a change in the request-type mix. This results in decreased performance, reduced QoS, and service disruptions. Some spikes are due to non-predictable events with unpredictable load volumes, while others occur due to predictable events with unpredictable load volumes [10].

Workloads that exhibit a significant number of workload bursts are known as *bursty workloads*. Because it is difficult to predict the intensity or the occurrence of a workload burst, bursty workloads create problems for cloud resource management (especially elasticity), since cloud providers have to host multiple applications with different workload behaviors. This creates problems for cloud auto-scaler algorithms. Some are better at dealing with periodic workloads, where previous behavior is a good indicator of future requirements. Others are better suited for dealing with bursty workloads with less predictable workload requirements [10].

3.4: Issues With Current Cloud Auto-Scaler Performance Evaluation Schemes

The current state of the art with respect to auto-scaler evaluation is immature due to a general lack of consistency between different research papers that makes it difficult to verify that the results observed in the literature are applicable to the real world. This is due to a number of factors:

- *A lack of consistent benchmarking metrics.* The issue of cloud auto-scaler evaluation is difficult to properly resolve due to the large number of different aspects to consider (quality of service [QoS], cost, etc.), and most extant metrics only measure a few aspects of the problem at a time [10].
- *The lack of formal, standard evaluation methodologies for auto-scaling algorithms*. The lack of widely accepted evaluation scenarios or standardized scoring metrics makes comparing different auto-scalers difficult, if not impossible [10]. Different papers in the literature use different metrics (such as response time, the auto-scaling demand index [ADI] [22], etc.), and different testing procedures (simulation vs. production cloud tests) for determining what constitutes a good cloud auto-scaler. What experimentation is carried out is limited to a handful of experiments that are difficult to generalize for, and are carried out not by comparing a given auto-scaler to other auto-scalers, but to pre-defined response times or static provisioning [10].
- *The use of only a few short workloads to characterize auto-scaler behavior.* Due to the lack of suitable, publicly available cloud workloads, much of the auto-scaling literature uses less than three real workload traces, often covering only a few seconds, minutes, or

days. This is problematic, both because the workload profile can dramatically affect the performance of a cloud auto-scaler and because the limited amount of workload traces used makes it impossible to draw general conclusions about the behavior of the auto-scalers being evaluated [10].

As a result, the overall lack of consistency between performance evaluation methods makes it difficult to compare the performance of any two different auto-scalers, because the research methodologies can vary wildly between papers and thus makes it difficult to determine whether the experimental results observed by the authors will actually hold up to real-world usage. Table 3.3 provides a list of examples from the literature.

Table 3.3: Survey of evaluation methods in the literature

Paper	**Experimental Platform**	**Performance Metric(s)**	**Workload source(s)**
WAC: A Workload analysis and classification tool for automatic selection of cloud auto-scaling methods [23]	Simulator	• Average Overprovisioning • Average Underprovisioning • Average number of oscillations	Mix of synthetic and real-world workloads
VM auto-scaling methods for high throughput computing on hybrid infrastructure [24]	Custom Testbed, Simulator	• Execution Time • Number of VMs • Percentage of tasks completed	Protein annotation workflow developed by the London e-Science Center
Using Application Data for SLA-aware Auto-scaling in Cloud Environments [25]	Simulator	• % out of SLA • CPU Load	Seven dumps of Twitter tweets from the 2013 FIFA Confederations Cup

Table 3.3: Continued

Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients [26]	Simulator	• Prediction error • UCSB metric • Cost	Animoto and seven IBM Cloud Application Traces
Optimal cloud resource auto-scaling for web applications [27]	Simulator, Production Cloud (Amazon EC2)	• Symmetrizing Kullback-Leibler (SKL) Divergence • RMSE • RSE • MAE • RAE • R^2 • Number of VMs • SLA violation rate	AOL, Sogou, and a real-world dataset from The University of Technology, Sidney (UTS)
Impact of user patience on auto-scaling resource capacity for cloud services [28]	Simulator	• Percentage of QoS violations • Allocated Resource Time • Percentage of user dissatisfaction • Aggregate request slowdown	Synthetic traces
Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers [29]	Custom Testbed	• Cost • Number of leased CPU cores • Cost/Invocations • SLA adherence	Synthetic traces
Evaluating the Auto Scaling Performance of Flexiscale and Amazon EC2 Clouds [30]	Production Cloud (Flexiscale, Amazon EC2)	Response time, average CPU utilization	Apache Jmeter-generated synthetic traces
Evaluating Auto-scaling Strategies for Cloud Computing Environments [22]	Simulator	Auto-scaling Demand Index (ADI)	Google cluster traces
Efficient autoscaling in the cloud using predictive models for workload forecasting [31]	Simulator	• Cost • Number of machines allocated	World Cup 1998 traces

Table 3.3: Continued

Dynamic Provisioning of Multi-tier Internet Applications [32]	Custom testbed	Response time	Rubis and Rubbos, using the World Cup 1998 traces
Cloud Performance Modeling with Benchmark Evaluation of Elastic Scaling Strategies [33]	Production Cloud (Amazon EC2, Rackspace)	Several, including execution time, efficiency, scalability, throughput, productivity, and scalability	BenchCloud, CloudSuite, Hi-Bench, TPC-W, YCSB
Automatic Resource Scaling for Medical Cyber-Physical Systems Running in Private Cloud Computing Architecture [34]	Custom testbed (OpenStack)	• Percentage of subtasks missing deadlines	Synthetic workload generated by medical CPS device simulators
Auto-scaling to minimize cost and meet application deadlines in cloud workflows [35]	Simulator	• Mechanism overhead • Utilization • Cost	Synthetic workload traces representing four workload patterns: • Stable • Growing • Cycle/Bursting • On-and-off
Application Performance Management in the Cloud using Learning, Optimization, and Control Rising adoption of cloud-based services [36]	Custom Testbed	• Response time	MongoDB, Rain

Table 3.3: Continued

An adaptive hybrid elasticity controller for cloud infrastructures [7]	Simulator (custom Python-based)	• S-: number of servers that the auto-scaler fails to provision on time per unit time • S+: Number of extra servers provisioned per unit time • Averaged value over time variants of the above	World Cup 1998 Traces
A tenant-based resource allocation model for scaling Software-as-a-Service applications over cloud computing infrastructures [37]	Custom Testbed Running a Tomcat-based SaaS platform on top of Eucalyptus	• Underutilization: Point of exhaustion (maximum useful payload that can be placed on a single VM without adversely affecting throughput as defined by number of requests / total time) • Overutilization: maximum number of VMs that can be released	Apache JMeter-generated synthetic traces
A cost-aware auto-scaling approach using the workload prediction in service clouds [38]	Simulator (CloudSim, modelling Amazon EC2 running various web-based services)	• Scaling costs • Resource usages • Resource utilization	Internally developed benchmarks modelling social networking services and video hosting services respectively

3.5: The PEAS Framework

In an attempt to address the issues with inconsistency in cloud auto-scaler performance evaluation, [10] proposed the Performance Evaluation Framework for Auto-Scaling (PEAS) is a framework for providing probabilistic guarantees on auto-scaler performance. It aims to provide easy-to-compare performance evaluations of cloud auto-scaling algorithms by providing formalized, standardized, and mathematically rigorous methodologies for providing probabilistic guarantees on auto-scaler performance based on the error from the ideal VM capacity outlay. It is based on scenario theory, a technique for providing approximate solutions to chance-constrained optimization problems in which instead of solving for an exact solution to a Chance-Constrained Optimization Problem (CCP) (which is NP-hard), one can solve for the best solution under the constraint that the probability that the actual value is less than or equal to that solution is greater than or equal to $1-\epsilon$, where ϵ is the probability that the actual distance value will exceed that of the calculated solution [10]. In terms of the MAPE loop, the PEAS framework simulates the *Analysis* and *Planning* phases.

3.5.1: Underlying Queuing Model

In order to develop the CCP, [10] models the cloud infrastructure as a G/G/N stable queue with a variable number of servers N (which represent VMs) operating in discrete time $k \in N$ as shown in Figure 1. An elasticity controller attempts to match the current number of servers $y(k)$ such that it satisfies the resource demand brought about by the incoming number of requests $\lambda(k)$, the number of queued requests that have yet to be serviced $q(k)$, the average service rate per server (in number of requests per time unit) r_{app}, and the required capacity to service long-running requests that require more than 1 time unit to complete C_{lrr}. The ideal number of servers required to meet all currently running requests y°(k) can be modeled by the following equation:

$$y^{\circ}(k) = \left\lceil \frac{\lambda(k)+q(k)}{r_{app}} \right\rceil + C_{lrr}(k) \tag{1}$$

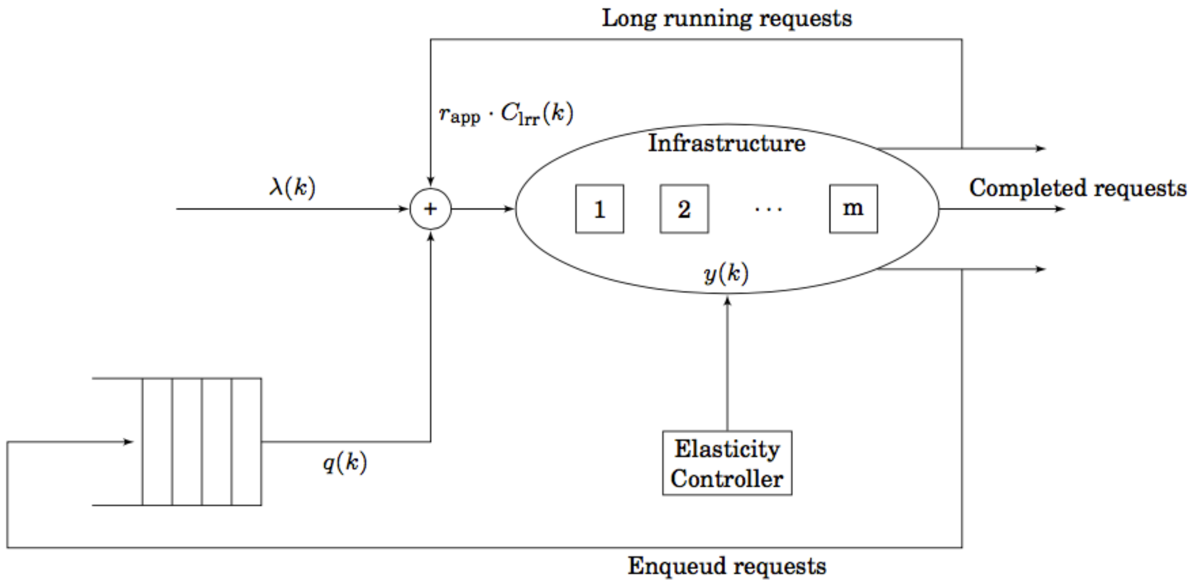

Figure 3.2: Queuing model for a service deployed in the cloud [10]

3.5.2: Formulation of the Chance-Constrained Optimization Problem (CCP)

[10] formulates the CCP for evaluating the goodness of an auto-scaler in PEAS in terms of the following conditions:

$$\text{CCP:} \ \min_{\rho} \rho$$

$$\text{subject to: } \mathrm{P}\{\mathrm{d}_{\mathcal{T}}(\mathrm{y}, \mathrm{y}^{\circ}) \leq \rho\} \geq 1 - \epsilon, \tag{2}$$

where ρ is the probabilistic solution to the CCP and $\mathrm{d}_{\mathcal{T}}(\mathrm{y}, \mathrm{y}^{\circ})$ is a distance function that represents the difference in behavior between the time series representing the actual capacity y and ideal capacity y°. This distance function is used as the performance metric to be considered by the PEAS algorithm, and any function that matches that signature can be used.

3.5.3: The PEAS Algorithm

To solve for the aforementioned CCP, PEAS runs N experiments using N different time series, each of which represents a different representation of the stochastic input, in order to generate the time series traces $y^{(i)}$ and $y^{\circ(i)}$ for each experiment before generating the distance values, filtering out the κ largest distance values from the set, and returning the largest remaining value in the set. The exact process for doing so is as follows [10]:

1. Get N different time series $\lambda^{(i)}(k),\ k = 1,\ 2,\ \ldots, |\mathcal{T}|,\ i = 1,\ 2,\ \ldots,\ N$ to use with the auto-scaling algorithms and let $\kappa = \lfloor \eta N \rfloor$.
2. Run the auto-scaling algorithms against each time series to generate $y(k)$ and $y^{\circ}(k)$ for each time series.
3. Compute the distance values $\hat{\rho}^{(i)} \coloneqq d_{\mathcal{T}}(y^{(i)},\ y^{\circ(i)})$ for each time series $i = 1,\ 2,\ \ldots,\ N$.
4. Determine the indices in the set of all input time series ($\{h_1, h_2, \ldots,\ h_\kappa\} \subset \{1,\ 2,\ \ldots,\ N\}$) of the κ largest values of the set of all distance values $\left\{\hat{\rho}^{(i)},\ i = 1,\ 2,\ \ldots,\ N\right\}$
5. Return the largest value of $\hat{\rho}^{(i)}$ from the set of indices that are not in the κ largest values of the distance values set ($\hat{\rho}^{*} = \max_{i \in \{1,\ 2,\ \ldots,\ N\} \setminus \{h_1,\ h_2,\ \ldots,\ h_\kappa\}} \hat{\rho}^{(i)}$)

The number of experiments N and the number of results to disregard κ are in turn affected by the empirical violation parameter η and the confidence parameter β. κ is set by the following equation:

$$\kappa = \lfloor \eta N \rfloor \tag{3}$$

β, in turn, is used as part of solving the following theorem along with η to produce a value for N:

If N is such that

$\sum_{i=0}^{\lfloor \eta N \rfloor} \binom{N}{i} \epsilon^i (1-\epsilon)^{N-i} \leq 1-\beta,$

Then the solution to the PEAS algorithm satisfies the restriction

$$\mathbb{P}\{d_{\mathcal{T}}(y,\, y^{\circ}) \leq \hat{\rho}^{*}\} \geq 1-\epsilon \qquad (4)$$

The following guidelines hold when choosing values of β, η, and ϵ:

- Increased η tends to result in increased N, as N scales as $\frac{1}{\epsilon-\eta}$, so the value of η depends on the desired value of N.
- β controls the probability that the size of the violation set will be larger than ϵ. Since the value of N is logarithmically proportional to $\frac{1}{\beta}$, β can be set as low as 10^{-10} without a significant increase in N.
- ϵ has a significant impact on the number of time series to run the algorithm with.

CHAPTER FOUR: RESEARCH METHODOLOGY

4.1: Experimental Parameters

The experiments that are the subject of this study were run using a simulator written in Python that was derived from that used in [10] with the exception of a change in the request processing code to address a bug that caused the number of queued requests to fall below zero. A custom Python analysis script was used to run the PEAS post-simulation analysis algorithm described earlier. The following simulation parameters from [10] are used for the PEAS algorithm:

- $\eta = 0.01$
- $\epsilon = 0.05$
- $\beta = 10^{-10}$
- $N = 796$
- $\kappa = \lfloor \eta N \rfloor = 7$
- Time unit length = 1 minute

Two test cases are considered in this study in which the average service rate r_{app} is modified to represent different classes of workloads. The first test case sets r_{app} to 22 requests/second (1.320×10^3 requests/minute) to mirror the service rate used in [10], which was derived from experimental results using the C-Mart cloud benchmark. The second test case sets r_{app} to 9.075×10^{-4} requests/second = 5.445×10^{-2} requests/minute, which was derived by taking the inverse of the average runtime of each job in the Google cluster traces [39]. Using a lower service rate allows us to better model the types of HPC workloads that are prevalent in the newer workload mixes that have been introduced in this experiment, as HPC workloads tend to have much longer per-request runtimes than the web requests that were modeled in [10]. To account for the small ($r_{app} < 0$) non-integral service rate parameters used in this experiment, the simulator used in this study was modified to use an exponential distribution instead of a Poisson distribution.

4.2: Workloads Used

In order to test the effects of workload mix on the performance of each auto-scaler, three sets of workload traces of request rates were used. Except for the Wikipedia traces, each trace was converted to the PEAS simulator format by taking the number of jobs that were submitted during each minute-long period of the trace. The additional traces are as follows:

- *ATLAS:* The ATLAS workload set is a set of 796 minute-by-minute traces derived from Standard Workload Format (SWF) traces that were created by the workload generator described in [40]. This workload generator uses a data model derived from the workload behavior of high-performance scientific computing jobs that were run on the Nordic Data Grid Facility (NDGF) as a part of the ATLAS project at the European Organization for Nuclear Research (CERN).
- *Wikipedia:* The Wikipedia workload set is a set of 796 hourly workload traces obtained from publicly available workload traces hosted by the Wikipedia foundation and used in [10]. It was generously provided by the original authors. The hourly request rates in the traces were scaled down to per-minute rates by dividing each entry by 60.
- *Production:* The Production workload set is a set of 796 per-minute parallel computing workload traces, each of which is 3.6 weeks long, that were derived from various real cluster and grid computing workload traces. They are intended to represent typical HPC and cloud computing workloads. They include every cleaned Standard Workload Format (SWF) workload trace as of 2016 at the Parallel Workloads Archive [41], six grid

workload traces in Grid Workload Format (GWF) from the Grid Workload Archives [42], the Google cluster traces [43], scientific computing workload traces from the CERIT Scientific Cloud [44], Hadoop traces from the OpenCloud research cluster at Carnegie Mellon University [45], and a workload trace obtained from a five month long trace at the Zewura cluster that were provided by the Czech National Grid Infrastructure MetaCentrum [44]. Because PEAS only accounts for the arrival times of each request when processing workload traces and models the individual request runtimes through the service rate, the converted traces only account for the per-minute arrival rates. A complete list of all workloads used under this set is listed in Table 4.1.

Table 4.1: List of workload traces included in the Production workload set

The NASA Ames iPSC/860 log [41]	Part of the Parallel Workload Archive, this is a 3-month long workload trace of interactive and batch jobs, mainly computational aerospace applications, that were run on the 128-node iPSC/860 hypercube at the Numerical Aerodynamic Simulation (NAS) Systems Division at NASA Ames Research Center.
The Los Alamos National Lab (LANL) CM-5 log [41]	Part of the Parallel Workloads Archive, this is a 2-year long workload trace from a 1024-node CM-5 system at Los Alamos National Lab.
The San-Diego Supercomputer Center (SDSC) Paragon log [41]	Part of the Parallel Workload Archive, this log consists of a two-year long workload trace from the 416-node Intel Paragon system at the San Diego Supercomputer Center (SDSC).
The Cornell Theory Center (CTC) IBM SP2 log [41]	Part of the Parallel Workload Archive, this log consists of an 11-month log workload trace from the 512-node IBM SP2 system at the Cornell Theory Center.
The Lawrence Livermore National Lab (LLNL) T3D log [41]	Part of the Parallel Workload Archive, this log consists of a 4-month long workload trace from the 256-node Cray T3D system at the Lawrence Livermore National Lab (LLNL).
The Swedish Royal Institute of Technology (KTH) IBM SP2 log [41]	Part of the Parallel Workload Archive, this log consists of an 11-month long trace from the 100-node IBM SP2 system at the Swedish Royal Institute of Technology (KTH) in Stockholm.

Table 4.1: Continued

The San Diego Supercomputer Center (SDSC) SP2 log [41]	Part of the Parallel Workload Archive, this log consists of a two-year long workload trace from the 128-node IBM SP2 system at the San Diego Supercomputer Center (SDSC).
The LANL Origin 2000 Cluster (Nirvana) log [41]	Part of the Parallel Workload Archive, this log consists of a 4-month long workload trace from a cluster of 16 Origin 2000 machines with 128 processors each at Los Alamos National Lab (LANL).
The OSC Linux Cluster log [41]	Part of the Parallel Workload Archive, this log consists of 22 months of workload data from a Linux cluster running at the Ohio Supercomputing Center (OSC)
The San Diego Supercomputer Center (SDSC) Blue Horizon log [41]	Part of the Parallel Workload Archive, this log consists of over two years of workload traces from a 144-node IBM SP system at the San Diego Supercomputer Center (SDSC).
The Sandia/Ross log [41]	Part of the Parallel Workload Archive, this log consists of a 3-year long workload trace from the Sandia Ross cluster, part of the Sandia CPlant project.
The HPC2N Seth log [41]	Part of the Parallel Workload Archive, this log consists of a 3.5-year long workload trace from a 120-node Linux cluster that was part of the High Performance Computing Center North (HPC2N), a joint operation in Sweden involving several universities and facilities.
The DAS2 5-Cluster Grid Logs [41]	Part of the Parallel Workload Archive, these one year logs cover workloads produced by parallel and distributed computing research communities at five different universities in the Netherlands, forming a grid consisting of five clusters.
The San Diego Supercomputer Center (SDSC) DataStar log [41]	Part of the Parallel Workload Archive, this log covers a year-long period of workload activity from the San Diego Supercomputer Center (SDSC) that was originally available from the NPACI JOBLOG repository.
The LPC Log [41]	Part of the Parallel Workload Archive, this log consists of a 10-month long workload trace from Laboratoire de Physique Corpusculaire (LPC) at Université Blaise-Pascal, Clermont-Ferrand, France. LPC was a cluster that was part of the EGEE project (Enabling Grids for E-science in Europe), which aimed to develop a grid infrastructure that could handle and analyze the data generated by the Large Hadron Collider (LHC).
The LCG Grid log [41]	Part of the Parallel Workload Archive, this log consists of an 11-day workload trace from multiple nodes of the Large Hadron Collider Computing Grid (LCG).
The SHARCNET log [41]	Part of the Parallel Workload Archive, this log consists of a 1-year long workload trace from the SHARCNET clusters located at Ontario, Canada.

Table 4.1: Continued

The LLNL uBGL log [41]	Part of the Parallel Workload Archive, this log consists of several months' worth of workload traces from a small BlueGene/L system at LLNL.
The LLNL Atlas log [41]	Part of the Parallel Workload Archive, this log consists of several months' worth of workload traces from a large Linux cluster called Atlas at LLNL that was intended for running large parallel jobs that cannot run on smaller computers.
The LLNL Thunder log [41]	Part of the Parallel Workload Archive, this log consists of several months' worth of workload traces from a large Linux Cluster called Thunder at LLNL that was intended for running large numbers of smaller jobs.
The ANL Intrepid log [41]	Part of the Parallel Workload Archive, this log consists of a 9-month long trace from a large Blue Gene/P system called Intrepid at Argonne National Laboratory that is used for scientific and engineering computing workloads.
The MetaCentrum log [41]	Part of the Parallel Workload Archive, this log consists of several months of workload traces from MetaCentrum, a national grid infrastructure at the Czech Republic. It was made available from [44].
The Potsdam Institute for Climate Impact Research (PIK) IBM iDataPlex Cluster log [41]	Part of the Parallel Workload Archive, this log contains over 3 years of workload traces from the 320-node IBM iDataPlex cluster at the Potsdam Institute for Climate Impact Research (PIK).
The CEA Curie log [41]	Part of the Parallel Workload Archive, this log consists of over 20 months of workload traces from the Curie supercomputer operated by CEA, a French government-funded technological research organization.
The Intel Netbatch logs [41]	Part of the Parallel Workload Archive, this log consists of four 1-month-long traces from the Intel Netbatch grid, each from a different cluster within Intel.
The University of Luxemburg Gaia Cluster log [41]	Part of the Parallel Workload Archive, this log consists of a 3-month long workload trace from the University of Luxemburg's Gaia cluster. It consists mainly of scientific workloads dealing with large data problems and engineering workloads dealing with physical simulations.
The MetaCentrum 2 log [41]	Part of the Parallel Workload Archive, this log consists of over 2 years of workload traces from MetaCentrum, a national grid infrastructure at the Czech Republic. It was made available from [44].
GWA-T-1 DAS2 [42]	Part of the Grid Workload Archive, this log consists of two years of workload traces from the DAS-2 system that were donated by the Advanced School for Computing and Imaging.
GWA-T-2 Grid5000 [42]	Part of the Grid Workload Archive, this log consists of over two years of workload traces from Grid'5000, an experimental grid platform in France consisting of 15 clusters geographically distributed across 9 sites.

Table 4.1: Continued

GWA-T-3 NorduGrid [42]	Part of the Grid Workload Archive, this log consists of over 3 years of workload traces from NorduGrid, a production grid for academic researchers in Denmark, Estonia, Finland, Norway, Sweden, etc.
GWA-T-4 AuverGrid [42]	Part of the Grid Workload Archive, this log consists of one year of workload traces from AuverGrid, a production grid platform consisting of 5 clusters in the Auvergne region of France that mainly run workloads related to biomedical and high-energy physics scientific research.
GWA-T-10 SHARCNet [42]	Part of the Grid Workload Archive, this log consists of a 1-year long workload trace from the SHARCNET clusters located at Ontario, Canada.
GWA-T-11 LCG [42]	Part of the Grid Workload Archive, this log consists of an 11-day workload trace from multiple nodes of the Large Hadron Collider Computing Grid (LCG).
***Zewura* Workload Log** [44]	This log covers 5 months of workload traces from the Zewura cluster, part of the Czech National Grid Infrastructure MetaCentrum.
Cerit-SC workload log (2013) [44]	This workload trace covers a 3-month period in 2013 from the CERIT Scientific Cloud (CERIT-SC), part of the Czech National Grid Infrastructure MetaCentrum.
Cerit-SC workload log (2015) [44]	This workload trace covers a 4-month period in 2015 from the CERIT Scientific Cloud (CERIT-SC), part of the Czech National Grid Infrastructure MetaCentrum.
Google ClusterData2011_2 Traces [39]	This workload trace covers 29 days of trace data from 2011 from a cluster of about 12.5k machines at Google.
CMU OpenCloud Traces [45]	These workload traces cover a 20-month period of activity from OpenCloud, a Hadoop cluster at Carnegie Mellon University (CMU) used for various academic, scientific, and engineering workloads.

4.3: Auto-scalers Used

The experiments described in this thesis use the following auto-scaler simulation code [10]. All auto-scaler algorithms are used with the default parameters provided by the simulation code, and are briefly described as follows:

- *React* [46]: *React* is a simple reactive dynamic scaling algorithm that uses threshold-based auto-scaling to add/remove VMs. It is one of the simplest scaling algorithms used in [10] and also the one that generated the best results.

- *Hist* [32]: *Hist* uses a histogram-based predictive technique that uses histograms of historical arrival rates to determine the number of resources to provision per hour. It also uses reactive provisioning to correct for prediction errors.
- *Adapt* [7]: *Adapt* adjusts the number of VMs based on both monitored load changes and predicted load changes. Predictions are based on the rate of change of the workload, and aims to adapt to sudden load changes while preventing premature resource release.

4.4: Distance Formulas Used

The following distance formulas [10] are listed as follows with a brief description. It should be noted that as these are all representing the distance between a desired value and the actual value, the lower the distance value the better:

- *Normalized Distance (d^{norm})*: The normalized distance penalizes under-provisioning and over-provisioning identically. It uses the squared 2-norm of the difference vector $\sum_{k\in\mathcal{T}} \|y^\circ(k) - y(k)\|^2$. To account for the difference in lengths between each time series, the normalization term $\frac{1}{|\mathcal{T}|}$ is introduced. It is represented by the following equation:

$$d_{\mathcal{T}}^{norm}(y,\, y^\circ) = \frac{1}{|\mathcal{T}|}\sum_{k\in\mathcal{T}} \|y^\circ(k) - y(k)\|^2 \tag{5}$$

- *Modified Hausdorff Distance (d^{sup})*: This distance metric is a modified Hausdorff distance that can account for the maximum discrepancy between the ideal and actual behavior and the probability that y will enter some set within the time horizon $\mathcal{T}$. It is represented by the following equation:

$$d_{\mathcal{T}}^{sup}(y,\, y^\circ) = \sup_{k\in\mathcal{T}} \|y^\circ(k) - y(k)\| \tag{6}$$

- *Over/Under-Provisioning (d^{over}/d^{under}):* These distance metrics account for the degree of over- and under-provisioning for a given auto-scaler, and are represented by the following equations:

$$d_{\mathcal{T}}^{over}(y,\ y^{\circ}) = \sup_{k\in\mathcal{T}} \|\max\{y(k) - y^{\circ}(k),\ 0\}\| \tag{7}$$

$$d_{\mathcal{T}}^{under}(y,\ y^{\circ}) = \sup_{k\in\mathcal{T}} \|\max\{y^{\circ}(k) - y(k),\ 0\}\| \tag{8}$$

- *OverT/UnderT (d^{overT}/d^{underT}):* The OverT and UnderT metrics are a modification of the over/under provisioning metrics that measures the average over- and under-provisioning in a time unit for a given auto-scaler, and are represented by the following equations:

$$d_{\mathcal{T}}^{overT}(y,\ y^{\circ}) = \frac{1}{|\mathcal{T}|}\sup_{k\in\mathcal{T}} \|\max\{y(k) - y^{\circ}(k),\ 0\}\|\Delta k \tag{9}$$

$$d_{\mathcal{T}}^{underT}(y,\ y^{\circ}) = \frac{1}{|\mathcal{T}|}\sup_{k\in\mathcal{T}} \|\max\{y^{\circ}(k) - y(k),\ 0\}\|\,\Delta k, \tag{10}$$

- *Adapted Auto-Scaling Demand Index (ADI):* The ADI is a measure representing the degree to which the auto-scaler is outside a given bound of acceptable utilization levels as represented by the parameters L and U, representing the lower and upper bound of acceptable utilization levels respectively. It is, in effect, a measure of the degree of over- and under-provisioning. A time-normalized version of ADI that accounts for time series length can be represented by the following equations:

$$u(k) = \frac{y(k)}{y^{\circ}(k)} \tag{11}$$

$$\sigma(k) = \begin{cases} L - u(k) & \text{if } u(k) \le L, \\ u(k) - U & \text{if } u(k) \ge U, \\ 0 & \text{otherwise.} \end{cases} \tag{12}$$

$$\sigma = \textstyle\sum_{k\in\mathcal{T}} \sigma(k) \tag{13}$$

$$d_{\mathcal{T}}^{ADI} = \sigma_{\mathcal{T}} = \frac{\sigma}{|\mathcal{T}|} \tag{14}$$

Because traces where no VMs are required to satisfy current demands ($y^{\circ}(k) = 0$) at some point in the trace can result in divide-by-zero errors with the original ADI algorithm, a modified version of the time-normalized ADI algorithm is used in which $u(k) = \frac{y(k)+1}{y^{\circ}(k)+1}$ in order to consider the degree of over- and under-utilization even when no VMs are required. This is referred to as the *zero-corrected ADI* metric in the rest of this thesis. Note that for the purposes of this experiment, L and U are set to 0.9 and 1.1 respectively.

CHAPTER FIVE: RESULTS AND DISCUSSION

5.1: Fast Service Rate ($r_{app} = 1.320 \times 10^3$ requests/minute)

This section describes the results of the experiment when it was conducted under conditions resembling a web server workload, with a relatively fast average service rate $r_{app} = 1.320 \times 10^3$ requests/minute. Note that the results for the Wikipedia workload set coupled with the fast service rate is likely to be the most accurate indicator of real-world performance, as the other two sets used are more representative of HPC workloads with plenty of long-running requests while the Wikipedia workload set represents web traffic to the Wikimedia family of websites.

5.1.1: Normalized Distance

Table 5.1 and Figure 5.1 show the solutions to the CCP for the normalized distance function described in (5) for each auto-scaling algorithm and workload set when running at the fast service rate. Based on these results, there is no clear winner as to which is the best overall auto-scaling algorithm. Under fast service rate conditions, the *React* algorithm shows the best overall ability to minimize allocation errors under the ATLAS workload set, but is the worst overall performer under the Wikipedia and Production workloads. The poor Wikipedia workload performance is of particular note here, as it is the one that is most representative of real-world usage. The *Hist* workload, by contrast, provides the best available performance on the Wikipedia set under fast service rates, and the *Adapt* workload provides the best available performance on the Production set.

Table 5.1: Normalized distance at fast service rate

Workload Set	React	Hist	Adapt
ATLAS	**1.684E+02**	4.206E+10	6.273E+03
Wikipedia	1.112E+49	**9.879E+14**	9.592E+15
Production	1.976E+07	2.228E+06	**2.203E+05**

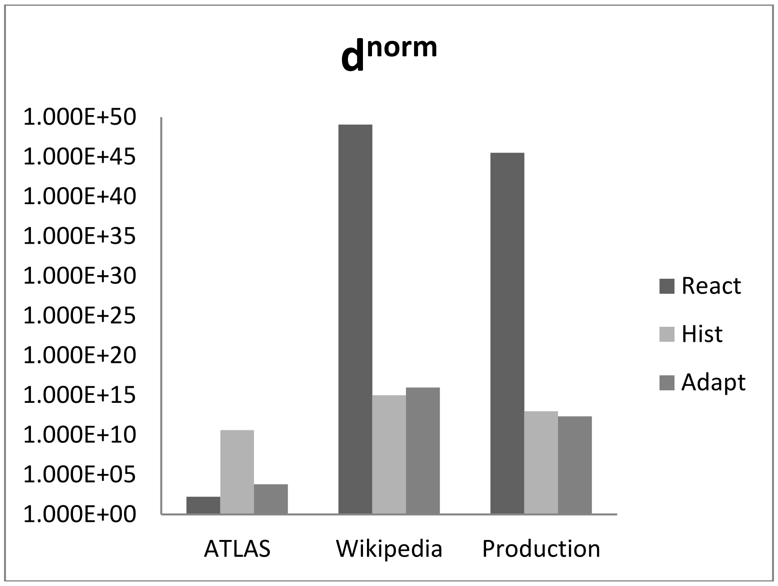

Figure 5.1: Normalized distance at fast service rate

5.1.2: Modified Hausdorff Distance

Table 5.2 and Figure 5.2 show the solutions to the CCP for the modified Hausdorff distance function described in (6) for each auto-scaling algorithm and workload set when using a fast service rate. The overall ordering of the performance under this metric mirrors that of the results for the normalized distance d^{norm}. The *Adapt* algorithm shows the best overall performance with the Production workload set, while the *Hist* algorithm does the best job of minimizing allocation errors under the Wikipedia workload set and the *React* algorithm does the best job under the ATLAS workload set.

Table 5.2: Modified Hausdorff distance at fast service rate

Workload Set	**React**	**Hist**	**Adapt**
ATLAS	**8.690E+02**	1.111E+07	6.582E+03
Wikipedia	5.518E+26	**4.286E+09**	2.185E+10
Production	8.338E+24	4.713E+08	**2.807E+08**

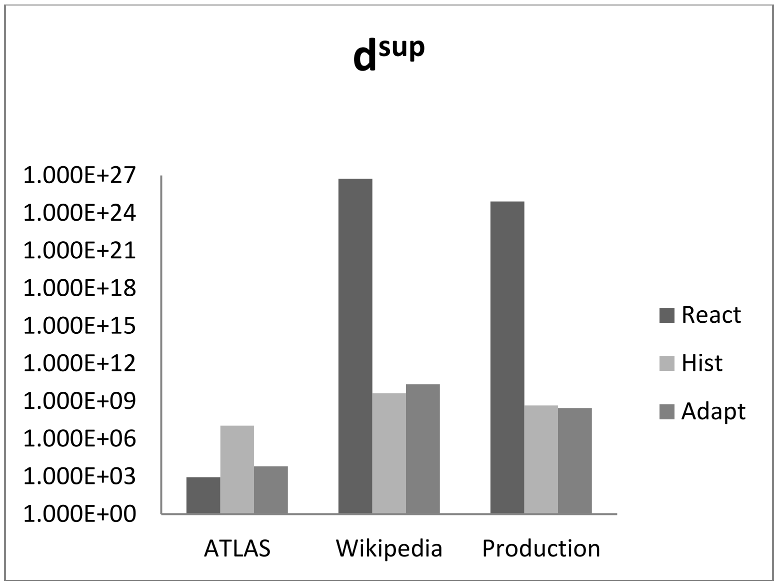

Figure 5.2: Modified Hausdorff distance at fast service rate

5.1.3: Over/Under-provisioning

Table 5.3 and Figure 5.3 show the solutions to the CCP for the over-provisioning metric using a fast service rate, while Table 5.4 and Figure 5.4 show the solutions to the CCP for the under-provisioning metric using a fast service rate. The *Hist* algorithm displays the best overall ability to minimize over-provisioning under all three workloads considered in the study while achieving the second-best levels of under-provisioning under the Wikipedia and Production workload sets and the worst under-provisioning under the ATLAS set. The *React* algorithm showed the second-best ability to minimize over-provisioning under the ATLAS and Production workload sets but showed poor under-provisioning and over-provisioning performance under the Wikipedia set and poor under-provisioning performance under the Production set. The *Adapt* algorithm showed the best overall ability to minimize under-provisioning across all workloads, but at a cost to over-provisioning; it provides the worst over-provisioning performance under the Production and Wikipedia sets but the second-best performance under the ATLAS set.

Table 5.3: Over-provisioning at fast service rate

Workload Set	React	Hist	Adapt
ATLAS	9.000E+00	**1.000E+00**	6.582E+03
Wikipedia	1.776E+25	**0.000E+00**	2.185E+10
Production	2.180E+02	**1.000E+00**	4.896E+03

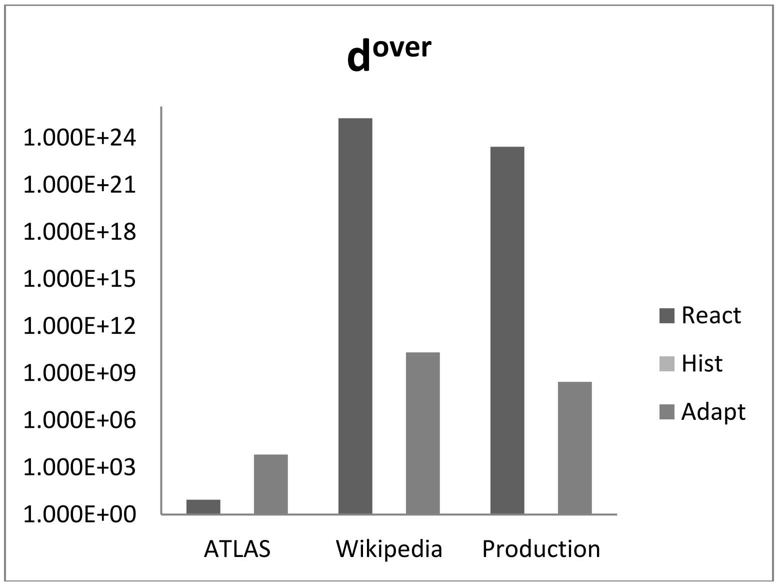

Figure 5.3: Over-provisioning at fast service rate

Table 5.4: Under-provisioning at fast service rate

Workload Set	React	Hist	Adapt
ATLAS	8.690E+02	1.111E+07	**8.390E+02**
Wikipedia	5.518E+26	4.286E+09	**6.334E+07**
Production	8.338E+24	4.713E+08	**8.150E+05**

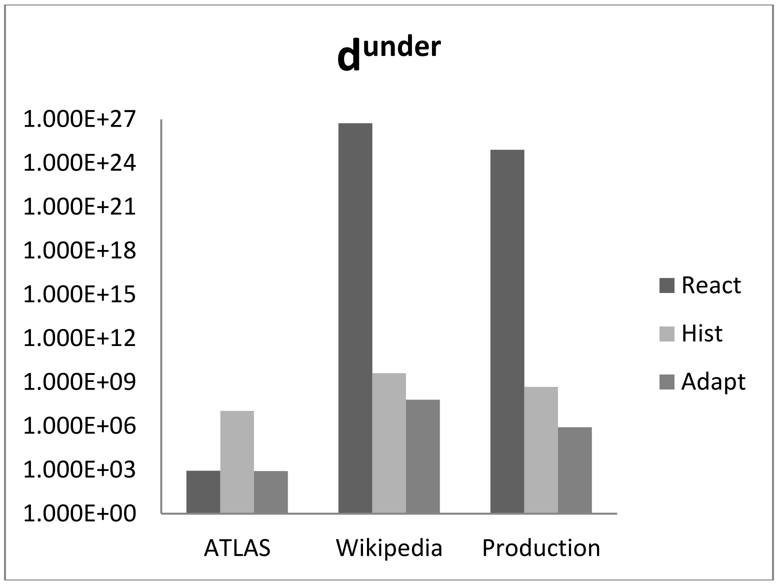

Figure 5.4: Under-provisioning at fast service rate

5.1.4: d^{overT}/d^{underT}

Table 5.5 and Figure 5.5 show the solutions to the CCP for the d^{overT} metric using a fast service rate, while Table 5.6 and Figure 5.6 show the solutions to the CCP for the d^{underT} provisioning metric using a fast service rate. Similar behavior was exhibited across all auto-scalers and workload sets between d^{overT}/d^{underT} and the over-provisioning/under-provisioning metrics respectively.

Table 5.5: OverT at fast service rate

Workload Set	**React**	**Hist**	**Adapt**
ATLAS	9.000E+00	**1.000E+00**	6.582E+03
Wikipedia	1.776E+25	**0.000E+00**	2.185E+10
Production	2.180E+02	**1.000E+00**	4.896E+03

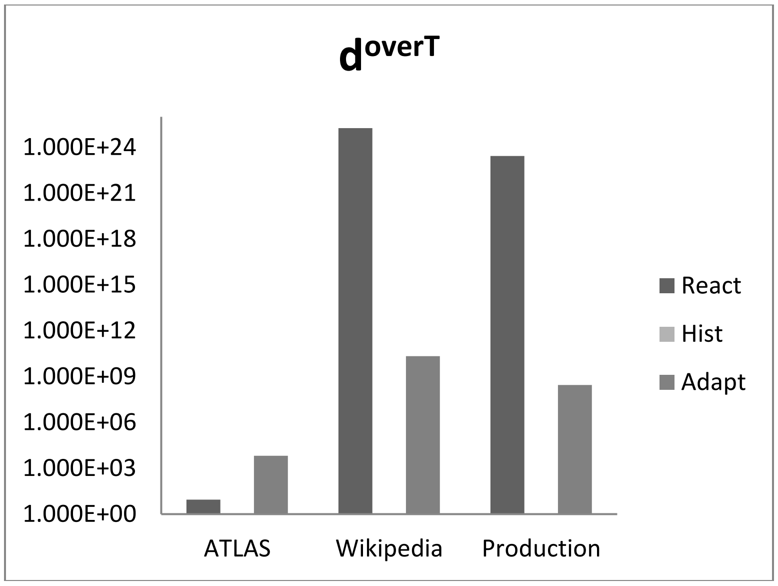

Figure 5.5: OverT at fast service rate

Table 5.6: UnderT at fast service rate

Workload Set	React	Hist	Adapt
ATLAS	1.684E+02	4.206E+10	**1.429E+02**
Wikipedia	1.111E+49	9.879E+14	**1.772E+11**
Production	1.976E+07	2.228E+06	**7.012E+04**

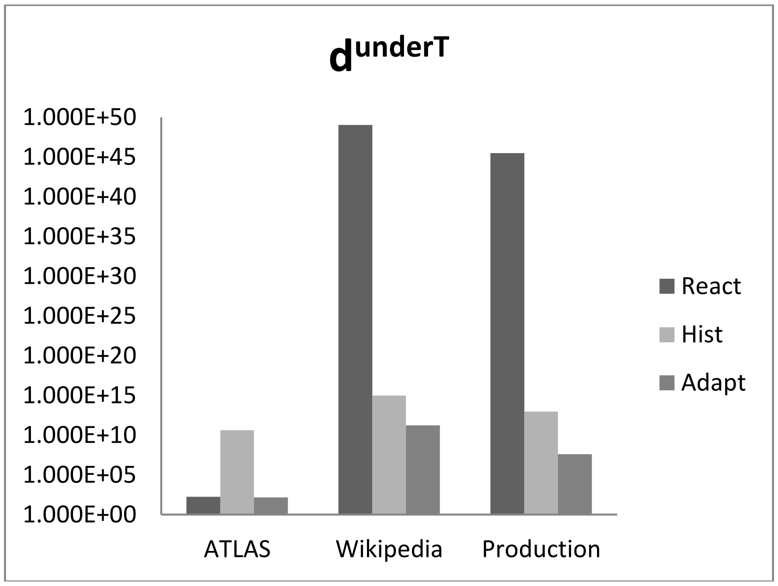

Figure 5.6: UnderT at fast service rate

5.1.5: ADI

Table 5.7 and Figure 5.7 show the CCP solutions for the zero-corrected ADI at fast service rates. Across all workload sets, the *React* algorithm demonstrates the best ability to closely track the allocation targets, although the *Hist* algorithm is second-best at tracking allocation targets. The *Adapt* algorithm demonstrates the worst overall performance.

Table 5.7: ADI (Zero-corrected) at fast service rate

Workload Set	**React**	**Hist**	**Adapt**
ATLAS	**2.827E-02**	7.395E-01	7.753E-01
Wikipedia	**3.037E-01**	8.999E-01	1.256E+02
Production	**2.799E-01**	9.000E-01	1.110E+01

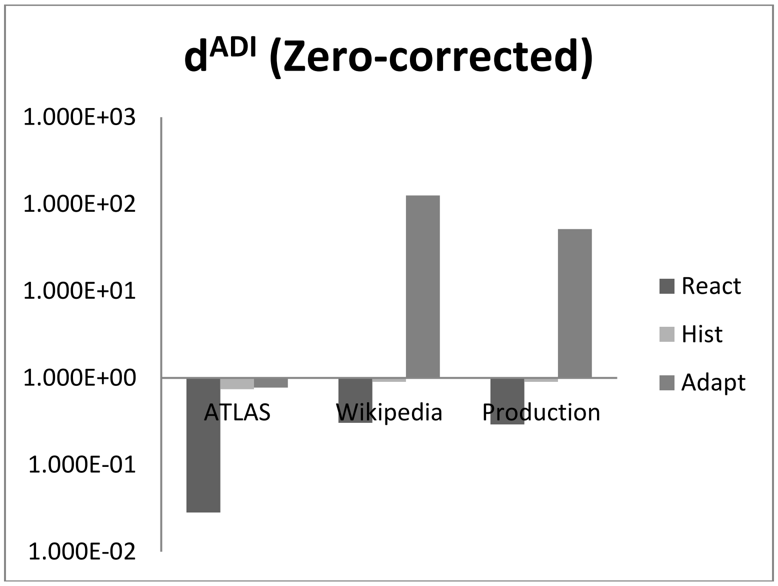

Figure 5.7: ADI (Zero-corrected) at fast service rate

5.2: Slow Service Rate ($r_{app} = 5.445 \times 10^{-2}$ requests/minute)

This section describes the results of the experiment when it was conducted under conditions representing high-performance computing workloads, with an average service rate of $r_{app} = 5.445 \times 10^{-2}$ requests/minute. Note that the results for the Production and ATLAS workload sets coupled with the slow service rate are likely to be the most accurate indicator of real-world performance, as the other two sets used are more representative of HPC workloads with plenty of long-running requests/jobs.

5.2.1: Normalized Distance

Table 5.8 and Figure 5.8 show the solutions to the CCP for the normalized distance function described in (5) for each auto-scaling algorithm and workload set. Under slow service rate conditions, the *Hist* algorithm shows the best overall ability to minimize allocation errors under the Wikipedia and Production workload mixes, while the Adapt algorithm provides the best performance under the ATLAS workload and the second-best performance under the other two workloads.

Table 5.8: Normalized distance at slow service rate

Workload Set	**React**	**Hist**	**Adapt**
ATLAS	1.612E+21	1.241E+10	**2.574E+08**
Wikipedia	1.347E+31	**1.001E+14**	2.783E+14
Production	3.152E+29	**4.977E+12**	1.801E+14

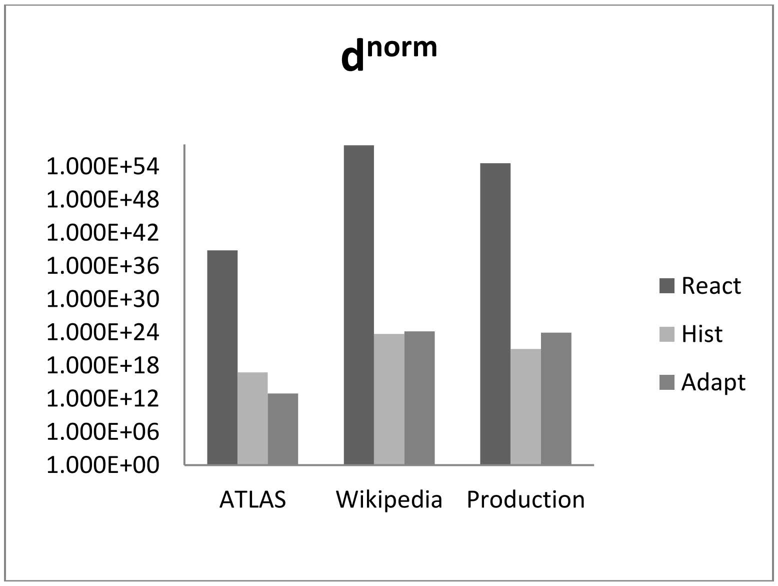

Figure 5.8: Normalized distance at slow service rate

5.2.2: Modified Hausdorff Distance

Table 5.9 and Figure 5.9 show the solutions to the CCP for the modified Hausdorff distance function described in (6) for each auto-scaling algorithm and workload set. As with the case under the fast service rate, the overall ordering of the performance under this metric mirrors that of the results for the normalized distance d^{norm}. The *Adapt* algorithm shows the best overall performance with the Production workload set, while the *Hist* algorithm does the best job of minimizing allocation errors under the Wikipedia workload set and the *React* algorithm does the best job under the ATLAS workload set.

Table 5.9: Modified Hausdorff distance at slow service rate

Workload Set	**React**	**Hist**	**Adapt**
ATLAS	1.612E+21	1.241E+10	**2.574E+08**
Wikipedia	1.347E+31	**1.001E+14**	2.783E+14
Production	3.152E+29	**4.977E+12**	1.801E+14

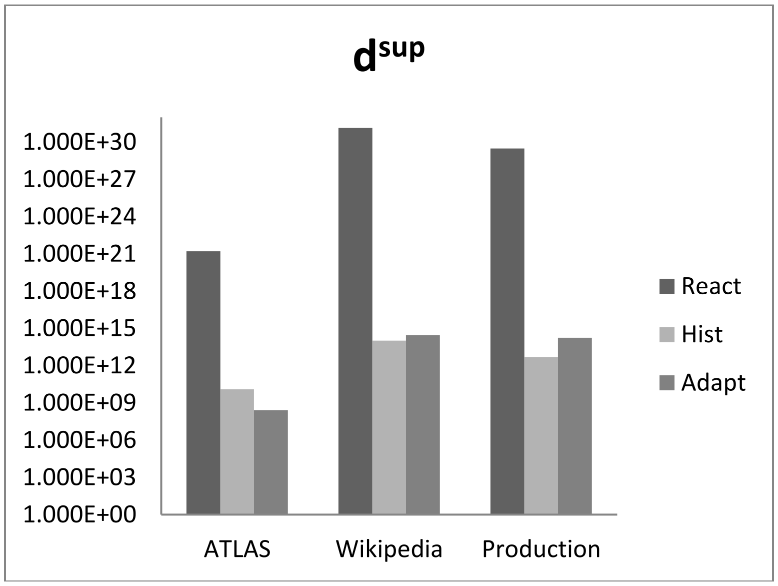

Figure 5.9: Modified Hausdorff distance at slow service rate

5.2.3: Over/Under-provisioning

Table 5.10 and Figure 5.10 show the solutions to the CCP for the over-provisioning metric, while Table 5.11 and Figure 5.11 show the solutions to the CCP for the under-provisioning metric. While the *Hist* algorithm does the best overall job of minimizing VM over-allocation while being the second-best overall at minimizing under-allocation, the *Adapt* algorithm does a better overall job of minimizing under-allocation while having the second-best overall performance at minimizing over-allocation. The *React* algorithm, by contrast, displayed poor overall performance under both metrics.

Table 5.10: Over-provisioning at slow service rate

Workload Set	**React**	**Hist**	**Adapt**
ATLAS	4.779E+19	**2.470E+02**	2.574E+08
Wikipedia	4.334E+29	**4.900E+01**	2.783E+14
Production	1.014E+28	**1.720E+03**	1.801E+14

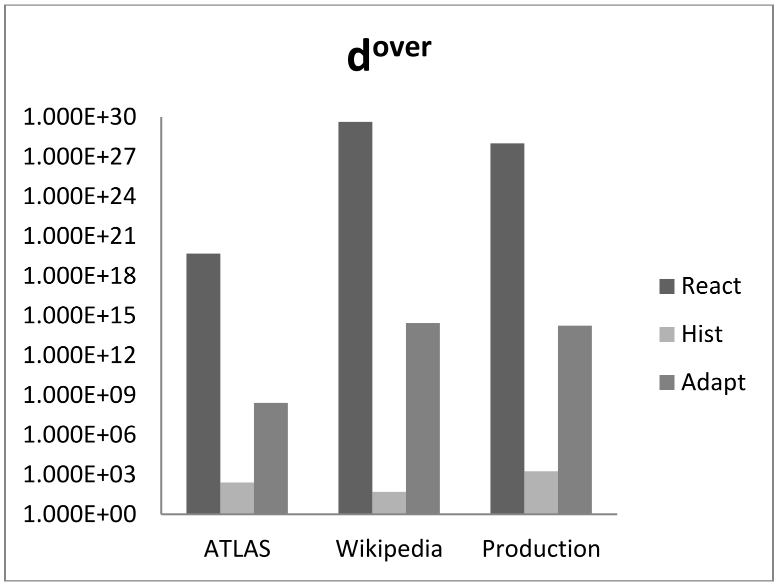

Figure 5.10: Over-provisioning at slow service rate

Table 5.11: Under-provisioning at slow service rate

Workload Set	**React**	**Hist**	**Adapt**
ATLAS	1.612E+21	1.241E+10	**2.988E+07**
Wikipedia	1.347E+31	1.001E+14	**8.914E+11**
Production	3.152E+29	4.977E+12	**1.666E+11**

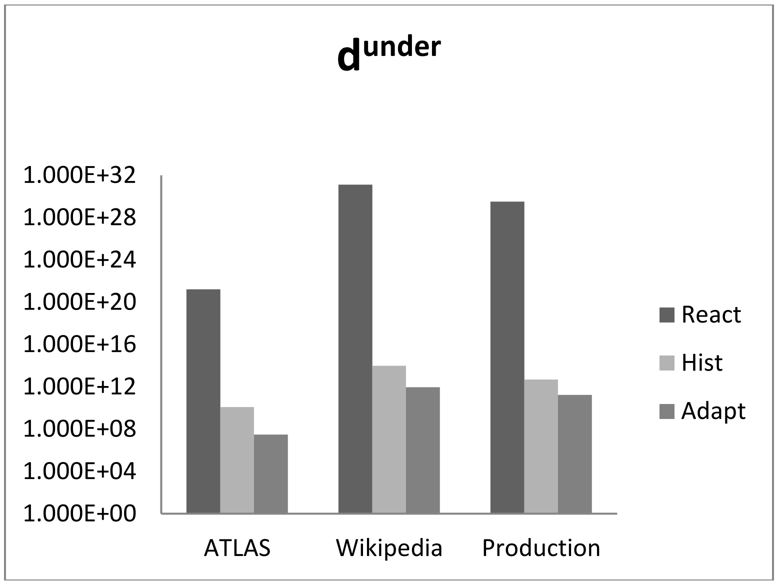

Figure 5.11: Under-provisioning at slow service rate

5.2.4: d^{overT}/d^{underT}

Table 5.12 and Figure 5.12 show the solutions to the CCP for the d^{overT} metric, while Table 5.13 and Figure 5.13 show the solutions to the CCP for the d^{underT} metric. As was the case with the fast service rate case, similar behavior was exhibited across all auto-scalers and workload sets between d^{overT}/d^{underT} and the over-provisioning/under-provisioning metrics respectively.

Table 5.12: OverT at slow service rate

Workload Set	**React**	**Hist**	**Adapt**
ATLAS	4.779E+19	**2.470E+02**	2.574E+08
Wikipedia	4.334E+29	**4.900E+01**	2.783E+14
Production	1.014E+28	**1.720E+03**	1.801E+14

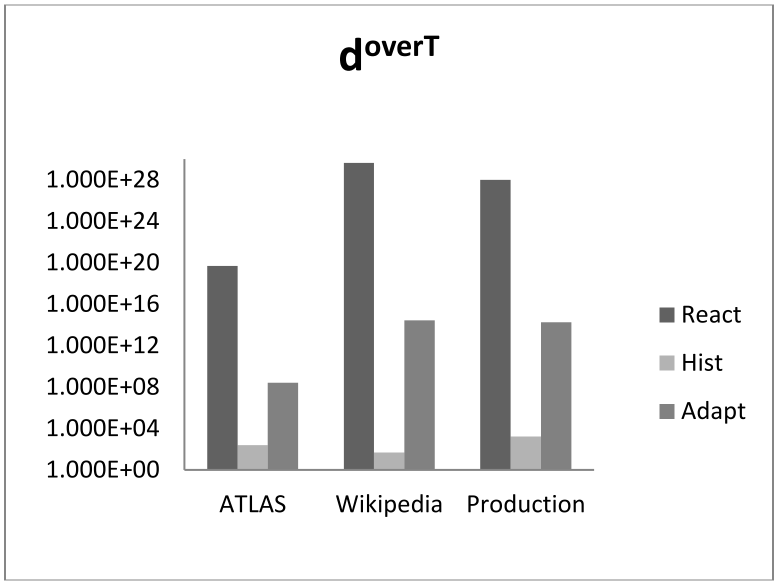

Figure 5.12: OverT at slow service rate

Table 5.13: UnderT at slow service rate

Workload Set	React	Hist	Adapt
ATLAS	6.874E+38	5.880E+16	**1.643E+11**
Wikipedia	6.644E+57	5.340E+23	**3.724E+19**
Production	3.953E+54	1.032E+21	**7.950E+17**

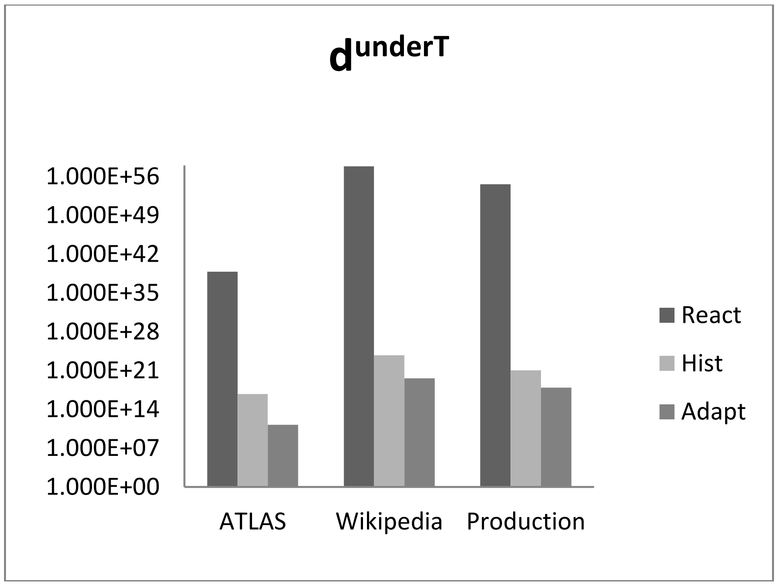

Figure 5.13: UnderT at slow service rate

5.2.5: ADI

Table 5.14 and Figure 5.14 show the CCP solutions for the zero-corrected ADI at slow service rates. A similar pattern to the fast service rate results in Table 5.7 and Figure 5.7 can be observed, with the *React* algorithm demonstrates the best ability to closely track the allocation targets, although the *Hist* algorithm is second-best at tracking allocation targets. The *Adapt* algorithm demonstrates the worst overall performance.

Table 5.14: ADI (Zero-corrected) at slow service rate

Workload Set	**React**	**Hist**	**Adapt**
ATLAS	**3.053E-01**	8.997E-01	1.497E+03
Wikipedia	**3.047E-01**	9.000E-01	4.044E+04
Production	**3.050E-01**	9.000E-01	4.586E+06

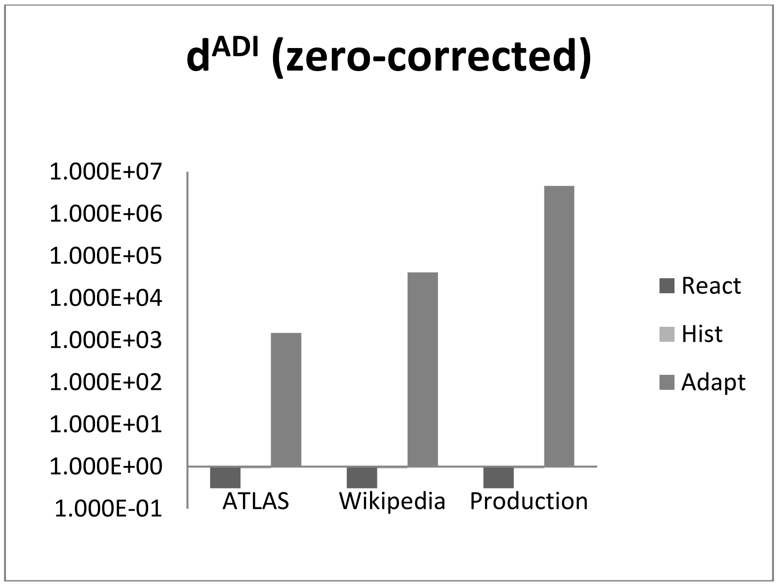

Figure 5.14: ADI (Zero-corrected) at slow service rate

CHAPTER SIX: CONCLUSION

In this thesis, the effects of modifying the workload mix and job runtimes (as represented by the service rate) on the accuracy of cloud resource allocation planning were studied. The divergent results observed between the fast service rate experiments and the slow service rate experiments for the d^{notm} and d^{sup} metrics illustrates the importance of tailoring auto-scaling strategies to the workloads and performance characteristics at hand, as well as the importance of considering multiple measures of performance when considering cloud auto-scalers. The surprisingly consistent results for the d^{over}/d^{under}, d^{overT}/d^{underT}, and d^{ADI} metrics across all workload sets and service rates suggest, however, that certain aspects of a given cloud auto-scaler's performance may well be consistent regardless of the workload mix.

The *Hist* algorithm does the best overall job of minimizing over-provisioning and thus minimizing resource waste; this comes at a cost of significantly higher under-provisioning and by extension a higher risk of failing to satisfy SLAs. Its ability to minimize allocation error in terms of d^{norm} and d^{sup} is highly workload-dependent, though. Under the fast service rate, it shows the best d^{norm} and d^{sup} performance under the Wikipedia workload set and second-rate or worse performance under the other two sets, which are not as representative for this service rate as the Wikipedia set and therefore not as critical. Under the slow service rate, it shows the best performance under the Wikipedia and Production sets and the second-best performance under the ATLAS set for the d^{norm} and d^{sup} metrics. Its d^{ADI} performance is second only to the *React* algorithm across all workloads and service rates.

Adapt, by contrast, trades off higher over-provisioning in exchange for lower under-provisioning. Its over-provisioning performance under fast and slow service rate conditions was second-best to worst across all workload sets, but its under-provisioning performance was the best across all workload sets and service rates. Under the fast service rate, it displays the best d^{norm} and d^{sup} performance under the Production set and the second-best performance under the Wikipedia and ATLAS sets. Similarly, under the slow service rate, it displays the best d^{norm} and d^{sup} performance under the ATLAS set and the second-best performance under the Wikipedia and Production sets. Its d^{ADI} performance, however, is consistently the worst across all workload sets.

Finally, *React* shows poor overall ability to minimize over-provisioning and under-provisioning across many of the workloads studied in this thesis. It shows the worst ability to minimize over-provisioning and under-provisioning across all workload sets in the slow service rate case, and under the fast service rate it is generally either the second-worst or worst-performing algorithm across all workload sets. It also shows the worst d^{norm} and d^{sup} performance under all workload sets under the slow service rate and all workload sets except ATLAS under the fast service rate. However, its d^{ADI} performance is better than the other two algorithms across all workload sets, suggesting that despite its poor worst-case allocation performance that it does a strong overall job of tracking the workload requirements within the bounds defined by the d^{ADI} metric.

As a next step towards extending the work covered in this study, more work needs to be done to characterize each of the workload sets in order to figure out the effects of specific workload characteristics on auto-scaler behavior. In addition, while this study only considers the performance of a given workload set under a fixed average service rate, future studies should consider tying the average service rate to each individual workload trace to better model the performance characteristics of a given trace (i.e. use lower service rates to model HPC workload traces while using higher service rates to model web server traffic).

REFERENCES

[1] K. Leochico and E. John, "Extending the Performance Evaluation Framework for Auto-Scaling (PEAS)," in *Proceedings of the 2016 International Conference on Grid, Cloud, & Cluster Computing*, Las Vegas, 2016.

[2] K. Leochico and E. John, "Evaluating the Performance Evaluation Framework for Auto-Scaling (PEAS) Under Alternative Workload Mixes," in *Proceedings of the 2017 International Conference on Grid, Cloud, & Cluster Computing*, Las Vegas, 2017.

[3] E. F. Coutinho, F. de Carvalho Sousa and Rubens, P. A. L. Rego, D. G. Gomes and J. de Souza and Neuman, "Elasticity in cloud computing: a survey," *annals of telecommunications - annales des télécommunications,* vol. 70, pp. 289-309, aug 2015.

[4] H. Hussain, S. U. R. Malik, A. Hameed, S. U. Khan, G. Bickler, N. Min-Allah, M. B. Qureshi, L. Zhang, W. Yongji, N. Ghani, J. Kolodziej, A. Y. Zomaya, C.-Z. Xu, P. Balaji, A. Vishnu, F. Pinel, J. E. Pecero, D. Kliazovich, P. Bouvry, H. Li, L. Wang, D. Chen and A. Rayes, "A survey on resource allocation in high performance distributed computing systems," *Parallel Computing,* vol. 39, pp. 709-736, nov 2013.

[5] D. Patterson, "Cloud computing and the RAD lab," in *Proc. Microsoft Research Cloud Futures Workshop (Redmond, WA, 2010), General Session Keynote, http://research.microsoft.com/en-us/events/cloudfutures2010/videos.aspx*, 2010.

[6] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson and A. Rabkin, "A view of cloud computing," *Communications of the ACM,* vol. 53, p. 50, apr 2010.

[7] A. Ali-Eldin, J. Tordsson and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *2012 IEEE Network Operations and Management Symposium*, 2012.

[8] N. R. Herbst, "Workload Classification and Forecasting," 2012.

[9] T. Lorido-Botran, J. Miguel-Alonso and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing,* vol. 12, pp. 559-592, dec 2014.

[10] A. Ali-Eldin Hassan, "Workload characterization, controller design and performance evaluation for cloud capacity autoscaling," Umeå University, 2015.

[11] IBM, *Get more out of cloud with a structured workload analysis,* 2011.

[12] P. Mell and T. Grance, "The NIST definition of cloud computing," Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 2011.

[13] S. S. Manvi and G. Krishna Shyam, "Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey," *Journal of Network and Computer Applications,* vol. 41, no. 1, pp. 424-440, May 2014.

[14] C. Vasquez, "Time Series Forecasting of Cloud Data Center Workloads for Dynamic Resource Provisioning," 2015.

[15] R. Moreno-Vozmediano, R. S. Montero and I. M. Llorente, "Key challenges in cloud computing: Enabling the future internet of services," *IEEE Internet Computing,* vol. 17, no. 4, pp. 18-25, 2013.

[16] A. Iosup, R. Prodan and D. Epema, "IaaS Cloud Benchmarking: Approaches, Challenges, and Experience," in *Cloud Computing for Data-Intensive Applications*, X. Li and J. Qiu, Eds., New, York: Springer New York, 2014, pp. 83-104.

[17] V. Stantchev, "Performance Evaluation of Cloud Computing Offerings," in *2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences*, 2009.

[18] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience,* vol. 41, no. 1, pp. 23-50, 2011.

[19] W. Mulia, J. Acken, D. Fritz, N. Sehgal, S. Sohoni and C. Stanberry, "Cloud Workload Characterization," *IETE Technical Review,* vol. 30, no. 5, p. 382, 2013.

[20] D. Karpenko, R. Vitenberg and A. L. Read, "ATLAS grid workload on NDGF resources: Analysis, modeling, and workload generation," *Future Generation Computer Systems,* vol. 47, pp. 31-47, 2015.

[21] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, New York, New York, USA, 2011.

[22] M. A. S. Netto, C. Cardonha, R. L. F. Cunha and M. D. Assuncao, "Evaluating Auto-scaling Strategies for Cloud Computing Environments," in *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, 2014.

[23] A. Ali-Eldin, J. Tordsson, E. Elmroth and M. Kihl., *WAC: A Workload analysis and classification tool for automatic selection of cloud auto-scaling methods,* 2015.

[24] J. Choi, Y. Ahn, S. Kim, Y. Kim and J. Choi, "VM auto-scaling methods for high throughput computing on hybrid infrastructure," *Cluster Computing,* vol. 18, pp. 1063-1073, sep 2015.

[25] A. A. D. P. Souza and M. A. S. Netto, "Using Application Data for SLA-aware Auto-scaling in Cloud Environments," p. 9, jun 2015.

[26] E. Caron, F. Desprez and A. Muresan, "Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients," *Journal of Grid Computing,* vol. 9, pp. 49-64, 2011.

[27] J. Jiang, J. Lu, G. Zhang and G. Long, "Optimal cloud resource auto-scaling for web applications," *Proceedings - 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013,* pp. 58-65, may 2013.

[28] M. de Assunção and Dias, C. H. Cardonha, M. A. S. Netto and R. L. F. Cunha, "Impact of user patience on auto-scaling resource capacity for cloud services," *Future Generation Computer Systems,* vol. 55, pp. 41-50, feb 2016.

[29] P. Hoenisch, I. Weber, S. Schulte, L. Zhu and A. Fekete, Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers, vol. 5900, L. Baresi, C. Chi and J. Suzuki, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 316-323.

[30] F. L. Ferraris, D. Franceschelli, M. P. Gioiosa, D. Lucia, D. Ardagna, E. Di Nitto and T. Sharif, "Evaluating the Auto Scaling Performance of Flexiscale and Amazon EC2 Clouds," in *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012.

[31] N. Roy, A. Dubey and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011,* pp. 500-507, 2011.

[32] B. Urgaonkar, P. Shenoy, A. Chandra and P. Goyal, "Dynamic Provisioning of Multi-tier Internet Applications," in *Second International Conference on Autonomic Computing (ICAC'05)*, 2005.

[33] K. Hwang, X. Bai, Y. Shi, M. Li, W.-G. Chen and Y. Wu, "Cloud Performance Modeling with Benchmark Evaluation of Elastic Scaling Strategies," *IEEE Transactions on Parallel and Distributed Systems,* vol. 27, pp. 130-143, jan 2016.

[34] Y. Ahn, A. Mo and K. Cheng, "Automatic Resource Scaling for Medical Cyber-Physical Systems Running in Private Cloud Computing Architecture," in *5th Workshop on Medical Cyber-Physical Systems*, 2014.

[35] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, New York, New York, USA, 2011.

[36] X. Zhu, "Application Performance Management in the Cloud using Learning , Optimization , and Control Rising adoption of cloud-based services," 2014.

[37] J. Espadas, A. Molina, G. Jiménez, M. Molina, R. Ramírez and D. Concha, "A tenant-based resource allocation model for scaling Software-as-a-Service applications over cloud computing infrastructures," *Future Generation Computer Systems,* vol. 29, pp. 273-286, jan 2013.

[38] J. Yang, C. Liu, Y. Shang, B. Cheng, Z. Mao, C. Liu, L. Niu and J. Chen, "A cost-aware auto-scaling approach using the workload prediction in service clouds," *Information Systems Frontiers,* vol. 16, pp. 7-18, 2013.

[39] J. Wilkes, "More Google cluster data," November 2011. [Online]. Available: http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[40] D. Karpenko, R. Vitenberg and A. L. Read, "ATLAS grid workload on NDGF resources: Analysis, modeling, and workload generation," *Future Generation Computer Systems,* vol. 47, pp. 31-47, 2015.

[41] D. Feitelson, "Logs of Real Parallel Workloads from Production Systems," 2015. [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

[42] "The Grid Workloads Archive," [Online]. Available: http://gwa.ewi.tudelft.nl.

[43] J. Wilkes, "More Google cluster data," [Online]. Available: http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[44] D. Klusáček, "Dalibor Klusáček - Home Page," 2016. [Online]. Available: http://www.fi.muni.cz/~xklusac/index.php?page=meta2009.

[45] K. Ren, "OpenCloud Hadoop cluster trace," [Online]. Available: http://ftp.pdl.cmu.edu/pub/datasets/hla/.

[46] T. C. Chieu, A. Mohindra, A. A. Karve and A. Segal, "Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment," in *2009 IEEE International Conference on e-Business Engineering*, 2009.

[47] R. J. Hyndman and A. B. Koehler, "Another look at measures of forecast accuracy," *International Journal of Forecasting,* vol. 22, pp. 679-688, oct 2006.

[48] D. G. Feitelson and B. Nitzberg, "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860," in *Job Scheduling Strategies for Parallel Processing*, 1995.

VITA

Kester Leochico is from Austin, TX. He earned a Bachelor’s degree in Electrical Engineering from The University of Texas-Pan American and is currently pursuing his Master’s degree in Computer Engineering from The University of Texas at San Antonio. His future plans include attending a Ph.D. program.