University of Moratuwa
Department of Computer Science & Engineering

CS4202 - Research and Development Project

inteliScaler

Workload and Resource Aware, Proactive Autoscaler for PaaS Cloud Frameworks

Group Members

110532R	Ridwan Shariffdeen
110375L	Tharindu Munasinghe
110056K	Janaka Bandara
110059X	Bhathiya Supun

Supervisors

Internal	Dr. H.M.N. Dilum Bandara
External	Mr. Lakmal Warusawithana, WSO2
	Dr. Srinath Perera, WSO2

Coordinated by
Dr. Malaka Walpola

THIS REPORT IS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF BACHELOR OF SCIENCE OF ENGINEERING AT UNIVERSITY OF MORATUWA, SRI LANKA.

February 21, 2016

Declaration

We, the project group inteliScaler hereby declare that except where specified reference is made to the work of others, the project inteliScaler - a resource & cost aware, proactive auto scaler for PaaS cloud is our own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgement.

Signatures of the candidates:

....
R.S. Shariffdeen [110532R]

....
D.T.S.P. Munasinghe [110375L]

....
U.K.J.U. Bandara [110056K]

....
H.S. Bhathiya [110059X]

Supervisor:

....

(Signature and Date)
Dr. H.M.N. Dilum Bandara

Coordinator:

....

(Signature and Date)
Dr. Malaka Walpola

Abstract

Cloud systems rely on virtualization techniques to provide computing resources on demand. Elasticity is a key feature where provisioning and de-provisioning of resources is done via auto-scaling. Reactive auto-scaling, in which the scaling actions take place just after meeting the triggering thresholds, suffers from several issues like risk of under provisioning at peak loads and over provisioning during other times. For example, in most of the Platform-as-a-Service (PaaS) frameworks, auto-scaling solutions are based on reactive, threshold-driven approaches. Such systems are incapable of catering to rapidly varying workloads, unless the associated thresholds are sufficiently low; however, maintaining low thresholds introduces the problem of over-provisioning under relatively stable workloads. Determining an optimum threshold, which solves these issues while minimizing the costs and meeting QoS demands is not a trivial task for a regular PaaS user. Alternatively, thresholds are not a good indication of QoS compliance, which is a key performance indicator of most cloud applications. Proactive auto scaling solutions, where future resource demands can be forecasted and necessary scaling actions enacted beforehand, can overcome these issues. Nevertheless, the effectiveness of such proactive scaling solutions depends on the accuracy of the prediction method(s) adopted.

We propose inteliScaler, a proactive and cost-aware auto-scaling solution to address the above issues by combining a predictive model, cost model, and a smart killing feature. First, we evaluated several forecasting models for their applicability in forecasting different workload patterns. Second, we introduce an ensemble workload prediction mechanism based on time series and machine learning techniques for making more accurate predictions on drastically different workload patterns. Simulation results using the RUBiS application and several empirical workload traces show significant cost savings by inteliScaler compared to a typical PaaS auto-scaler. Moreover, our ensemble method produces significantly lower forecast errors compared to the use of individual forecasting models and the prediction technique employed in Apache Stratos, an open source PaaS framework. We further evaluated the proposed approach by implementing it on Apache Stratos PaaS framework, and then deploying and testing it on AWS EC2. Results show significant QoS improvements and cost reductions relative to a typical reactive and threshold-based PaaS auto-scaling solution.

Acknowledgement

First and foremost we would like to express our sincere gratitude to our project supervisor, Dr. H.M.N. Dilum Bandara for the valuable guidance and dedicated involvement at every step throughout the process.

We would also like to thank our external supervisors Dr. Srinath Perera and Mr. Lakmal Warusawithana for the valuable advice and the direction given to us regarding the project.

We would also like to acknowledge the support given by Amazon Web Services for providing us an educational research grant, which was much helpful to do our research on a real cloud platform with no cost.

In addition, we would like to thank Mr. Lajanugen Logeswaran for the insights and valuable help given to us in the context of cloud scaling and performance testing.

We would like to express our warm gratitude to Dr. Malaka Walpola for coordinating the final year projects.

Last but not least, we would like to express our greatest gratitude to the Department of Computer Science and Engineering, University of Moratuwa for providing the support for us to successfully finish the project.

Contents

List of Figures

List of Tables

Abbreviations

ANN Artificial Neural Network

APE Absolute Percentage Error

ARIMA Auto Regressive Integrated Moving Average

ARMA Auto Regressive Moving Average

AWS Amazon Web Services

CEP Complex Event Processor

EC2 Elastic Computing Cloud

EWMA Exponentially Weighted Moving Average

GCE Google Compute Engine

IaaS Infrastructure as a Service

LA Load Average

MAPE Mean Absolute Percentage Error

MC Memory Consumption

NIST National Institute of Standards and Technology

PaaS Platform as a Service

QoS Quality of Service

RDS Relational Database Service

RIF Requests in Flight

RMSE Root Mean Square Error

RUBiS Rice University Bidding System

SaaS Software as a Service

SLA Service Level Agreement

VM Virtual Machine

WIPS Web Interactions Per Second

Chapter 1

Introduction

1.1 Background

Cloud platforms are becoming increasingly popular as a means of effortlessly hosting web applications and services with high availability, scalability and low cost. Cloud computing is a utility and service based computing model which enables access to a shared pool of configurable computing resources on demand [1]. These resources can be provisioned and released with minimal human interaction and management effort. Cloud platforms fall into three main categories, namely:

- SaaS (Software as a Service): readymade applications available for direct client use, e.g., Google Docs
- PaaS (Platform as a Service): application development and hosting platforms for cloud developers, e.g., Google App Engine
- IaaS (Infrastructure as a Service): low-level infrastructure (usually virtual machines) available on demand, e.g., Amazon EC2 (Elastic Compute Cloud)

PaaS provisioning is preferred by most cloud application developers, as it provides a simplified interface to the underlying resources and the availability of extra facilities such as extended development tool support. Generally, a PaaS is set up by encapsulating resources of an existing IaaS provider (e.g., Apache Stratos [2] on Amazon AWS [3]). This allows the PaaS to scale based on its resource requirements, simply by applying scaling decisions to the underlying IaaS. Multi-cloud support is also offered by some PaaS frameworks, which implies that a user can run applications on one of several cloud infrastructures simply by deploying it on such a PaaS.

Scalability is a major driving force of all cloud applications. This is often achieved via an *auto-scaling process*, which dynamically allocates and deallocates resources for a cloud application to optimize its resource utilization while minimizing the cost as well as achieving the desired Quality of Service (QoS) and availability goals [4][5]. These conflicting goals are typically defined via a Service Level Agreement (SLA).

Auto-scaling approaches fall into two main categories. In reactive, rule-based auto-scaling, users have to specify the policies that would evaluate when to trigger the scale-up or scale-down events, and scaling would occur only after the scaling event is triggered [6]. In proactive, workload-aware auto-scaling, the cloud scales itself via self-awareness of its future workload. The latter is suitable for most situations where workload behaviour cannot be captured into a finite set of user-defined rules, or when users lack the expertise on defining efficient auto-scaling policies.

1.2 Motivation

Research shows that there is still a need for better proactive autoscaling techniques, which could drastically reduce the inefficiency of resource allocation and minimize the unnecessary cost for the customer due to over-provisioning of resources, and potential losses due to under-provisioning. Especially at the PaaS level, existing auto-scaling solutions are mostly rule based, reactive systems which heavily depend on configurable parameters.

1.2.1 Proactive Auto-Scaling

Although policy-aware auto-scaling seems to be the most customizable approach from the users perspective [7], it has several weaknesses:

- Necessity of prior user expertise (e.g., of traffic patterns, service health patterns and IaaS/PaaS parameters like pricing models and packages) for composing effective rules that can carry out scaling efficiently
- Existing systems do not consider the startup time of an instance (fault tolerance time) when taking an auto-scaling decision
- Inability to adapt to workload patterns (e.g., thrashing during load fluctuations)

Although some alternatives like hierarchical rulesets have been introduced in some implementations, policy-based auto-scaling has failed to produce the level of flexibility required for real-time auto-scaling of production systems [6].

Researches have identified the proactive approach (workload prediction) to be the most promising alternative for handling the above issues [4]. Research is already underway on workload prediction, using techniques like time-series analysis, reinforced learning, queuing theory and control theory [4][6].

1.2.2 Elasticity at PaaS Level

There are existing auto-scaling implementations at both IaaS and PaaS levels. At IaaS level, fine-grained control is available for an auto-scaler, beyond the VM (container) level (e.g., adjusting of memory and storage allocation of a given VM) [8]. However, this has its own downside as it adds higher complexity to the problem of optimal resource allocation for each user. In addition, an IaaS provider has to fulfill two goals simultaneously, which further increases the complexity of the required auto-scaling solution:

- Optimizing usage of limited physical resources allocated for provisioning the virtual IaaS infrastructure
- Maximizing potential revenue from IaaS customers (e.g., serving one customer with excessive resource demand may be more profitable than serving several customers with small resource demands) [4]

As a result, most IaaS providers have less interest in integrating auto-scaling solutions in their platforms. Those who do (like AWS using Amazon Auto Scaler) typically use proprietary techniques which are not publicly available due to various reasons.

While IaaS auto-scalers can access lower-level cloud metrics (e.g., CPU usage of individual VMs against time), they are not application-aware. In contrast, an auto-scaler operating at a higher level can use application-aware metrics to produce better auto-scaling decisions. For example, in case of a LAMP stack, an IaaS auto-scaler will scale the MySQL database, PHP application and Apache server in a non-coordinated manner, while a PaaS auto-scaler will be able to scale them parallely with centralized control, to minimize conflicts and inconsistencies.

Further advantages of a PaaS-layer auto-scaler include the ability to benefit from higher-level features available in the cloud such as aggregated workload statistics and deployment topology, and compatibility across IaaS platforms in case the base PaaS is compatible with multiple IaaS platforms (e.g., Apache Stratos).

1.2.3 Optimizing Resource Allocation

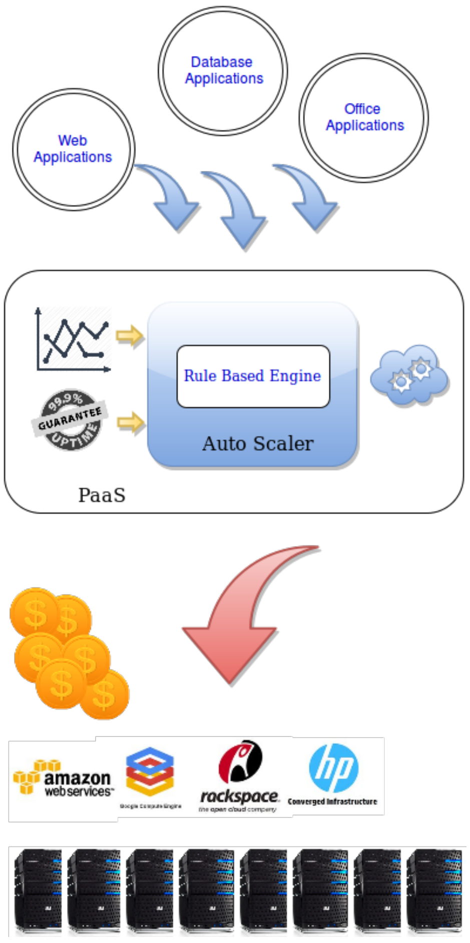


Figure 1.1: High-level overview of an existing PaaS auto-scaling solution.

Different application components in cloud environments have different resource requirements. For example, a storage management service may require a large volume of storage while a video streaming website may require high-bandwidth network capabilities. Going with such specialized requirements, cloud providers offer resource packages with specific capabilities, such as VMs with high CPU power and less memory suitable for CPU-intensive tasks. However, as shown in Figure 1.1 a typical auto-scaler at Paas level do not harness the benefits of matching such specializations to workload requirements, and naively pick a generic set of resources each time a scale-up is required.

In addition, different IaaS providers offer different pricing models, which leads to the possibility of using beneficial resource usage schemes like smart kill wherein a VM is not terminated until the end of its current pricing time unit (e.g., in case of a hourly billing scheme, until the end of a full billing hour). Although some auto-scalers have accompanied such features in support of specific IaaS platforms (e.g., AppScale auto-scaler [8] when used on AWS), there is no generic model for capturing such options and making them available for PaaS-level auto-scaling [7].

1.2.4 PaaS Framework Selection

Using detailed application and hardware metrics, an auto-scaling solution at PaaS level can provide an optimized resource allocation best suited for the application workload and SLA. To demonstrate the effect of an PaaS auto-scaler with we would be basing our research on Apache Stratos [2], an open-source PaaS framework originally developed by WSO2 and currently managed by the Apache Software Foundation. It can be used to configure and deploy a PaaS on existing IaaS resources, and supports multiple IaaS providers, including AWS, OpenStack, Google Compute Engine and Microsoft Azure. Benefits of selecting Stratos for the research include [2]:

- modular nature of Stratos architecture, allowing easy modifications.
- availability of a mock IaaS as a component project for testing and evaluation.
- easiness of deploying legacy applications using cartridges ability to set up partitions to control IaaS resources abundance of support, from Apache Community and the WSO2 Stratos Team.

1.3 Problem Description

1.3.1 Goal

Auto-scale computing resources in a PaaS cloud environment based on the current workload and resource usage, while predicting the workload to reduce the cost and to meet the desired QoS and SLA goals.

The proposed solution will satisfy the desirable properties of cloud systems such as elasticity, on-demand service and minimum cost usage. The solution will consider three aspects which includes, service level agreement, workload data and the pricing scheme of the IaaS. We plan to implement our solution for the Apache Stratos PaaS because its current auto-scaler primarily focuses on rule-based auto-scaling, and there is significant room for improvement on the predictive aspect. As listed above there are many advantages of Stratos compared to other PaaS such as OpenShift [9] and Cloud Foundry. It can be used to work with multiple databases, many application servers, multiple programming languages and many legacy systems as well.

1.3.2 Objectives

Objective of this research include the following:

- Analysis of existing workloads to identify typical workload patterns
- Prediction of workloads (e.g., ranging from few minutes to an hour) based on current data and the patterns observed from the past data
- Analysing resource characteristics with respect to their capabilities (e.g., CPU, memory, and IO), time, delay and cost
- Improving the policy mechanism of Stratos to efficiently allocate resources
- Developing a cost model to capture the cost of resources, revenue for providing desired service(s), and penalty for not meeting SLA. Cost of resources will be dependant on pricing scheme offered by the underlying IaaS, such as reserved pool, reserved instances and sustained use policy.

- Integrating the predictor, resource analyzer and cost models into one controller with an application-specific model to provide optimized auto-scaling solution
- Working model to be evaluated for its performance using a simulation platform with real-world and synthetic workloads

1.3.3 Contributions

Contributions of this project include the following:

- Set of workload prediction algorithms which can predict different workload patterns with varying characteristics. The implementation will be an ensemble component which dynamically changes the algorithm in use according to the given characteristics of the application in use as well as the prediction error.
- Workload and cost aware resource optimizer
- Resource usage monitor
- Visualizing component

Currently PaaS systems are not fully aware of the variety of resources provided by different IaaS providers, and their pricing models. Therefore, when scaling up, they generally spin up instances of the same type, regardless of finer details of resource requirements. On the other hand, they do not sufficiently assess cost and availability concerns when scaling down. We propose to improve this mechanism by selecting resources based on application workload patterns, available resource types and pricing of resources. For example, allocating memory optimized instance would be cost effective for some application while another application may require high CPU but less memory. Also, scale-down mechanism can be improved by introducing features like smart kill. The result is more efficient resource allocation for handling the predicted load, resulting in lower costs for PaaS users. Further, we propose to implement graph-based views of predicted vs. actual workload and normal vs. optimized resource usage and cost predictions.

1.4 Outline

Auto-scaling for PaaS can be achieved using different approaches that have been reviewed in our work. The remainder of this document is organized as follows.

Chapter 2 consists of five sections, provides a detailed analysis of related work in auto-scaling. We first describe cloud auto-scaling in general, and move on to a review of existing PaaS auto-scaling solutions. Next we discuss existing workload prediction techniques including time series analysis and machine learning, and different resource allocation approaches from the viewpoints of quality of service and cost. Finally, we review performance evaluation approaches for cloud systems, considering workload generators, experimental workload data, and application benchmarks.

Chapter 2

Literature Review

Many researches have already covered the field of cloud computing. The goal of this literature review is to identify problems in cloud auto-scaling, research questions, and a thorough analysis of auto-scaling systems for the cloud. We first introduce the concept of cloud auto-scaling with respect to IaaS and PaaS, and then we look into the existing auto-scaling solutions available for PaaS systems, followed by a detailed analysis of prediction techniques, optimum resource allocation and performance evaluation methods.

2.1 Cloud Auto-Scaling

Cloud computing is a model for enabling ubiquitous, convenient, and on-demand network access to a shared pool of configurable computing resources (such as networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [1].

From the hardware perspective, this brings up three main characteristics [5]:

1. The illusion of infinite computing resources available on demand
2. The elimination of an up-front commitment by cloud users
3. The ability to pay for the use of computing resources on a short-term basis as needed

In summary, key advantages of the cloud computing model over traditional hosting models rely on its ability to allocate and deallocate resources based on the applications requirements. For a small application on the cloud, it might be possible for a human to allocate and deallocate resources based on manual provisioning. But for large-scale applications it is impossible to depend on scaling with manual provisioning. Even for small-scale applications, manual scaling would impose hindrances in gaining the true advantages of cloud computing. Therefore, dynamic resource allocation and deallocation methods, or auto-scaling solutions, are demanded and developed.

Based on the nature of resource allocation, auto-scaling can be classified as:

Vertical auto-scaling: allocating or deallocating resources on a single VM (Fig. 2.1)[10]. Also known as scaling up or down, it is usually implemented by changing the partitioning of resources. Although this is a faster auto-scaling option (as modern hypervisors support partition changes without instance restarts), vertical auto-scaling is limited by the capacity of individual resources available on the host system.

Horizontal auto-scaling: adjustment of the number of VM instances in order to scale in or out (Fig. 2.2)[10]. Even though this may result in migration time as well as considerable wastage of resources, horizontal auto-scaling solutions are generally more versatile [11]

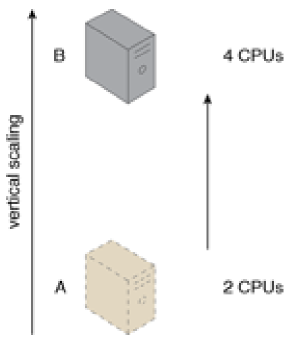


Figure 2.1: Vertical auto-scaling

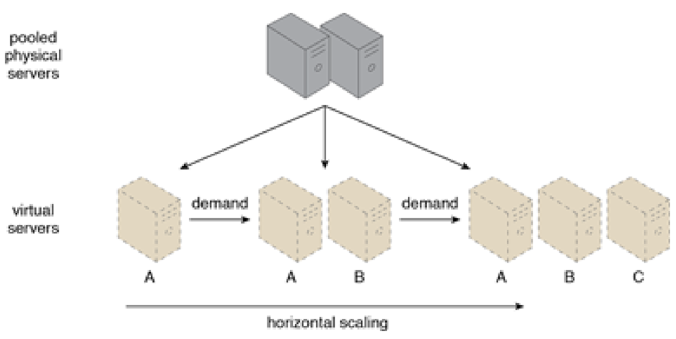


Figure 2.2: Horizontal auto-scaling

Due to scaling limitations on vertical auto-scaling, most of the auto-scaling solutions to date are based on the horizontal auto-scaling approach.

Auto-scaling solutions can also be categorized based on the type of provisioning:

Reactive auto-scaling: In these systems, auto-scaling conditions are predefined. When the conditions are met, a predefined scaling up or scaling-down action is triggered. A common implementation of the reactive approach is rule-based auto-scaling. This simple approach typically involves creating two rules to determine when to scale up and scale down. For example, if the CPU usage of VMs is over 70% a new instance is spun up in the auto-scaling group, while an instance is spun down when the CPU usage is less than 40%.

Proactive auto-scaling: In proactive or predictive auto-scaling, the system tries to anticipate the future load and schedule required resources ahead of time. Time series analysis is one of the most common approaches used to predict the future load in advance.

A good auto-scaling solution would use a hybrid approach, as predictive methods would be good for systems with periodic workload patterns while reactive methods can be used to encounter unexpected variations (e.g., spikes) in the workload.

Auto-scaling solutions can also be categorized based on the layers at which they are implemented, more specially the IaaS and PaaS layers.

2.1.1 IaaS Auto-Scaling

IaaS providers lease computational resources such as virtual machines (VMs) and storage services of different types, under a given pricing model. The application provider or the IaaS consumer tries to reduce the cost incurred while achieving the desired level of QoS (quality of service) [12]. To achieve this, most IaaS providers provide mechanisms that allow users to set up dynamic allocation for resources. AWS Auto Scaling and Azure Auto-scaling are two such solutions.

Almost all the solutions currently provided by IaaS providers rely on rule based approach. Here, two main actions are expected from the user:

- Creating auto-scaling groups
- Defining rules with threshold values

While these types of reactive, rule-based approaches are useful to a considerable extent, big consumers are not entirely satisfied with such solutions. For example, the online content provider Netflix, one of the biggest consumers of AWS, has built its own auto-scaling engine Scryer to address the challenges of cloud auto-scaling [13].

2.1.2 PaaS Auto-Scaling

When an application developer decides to use a PaaS over an IaaS, responsibility of auto-scaling is moved from the cloud application developer to the PaaS provider. For example, consider an e-commerce application that uses the LAMP stack. If the system administrators are to maintain and auto-scale this application in an IaaS, it would require them to become experts in the application itself, as well as in the IaaS on which it runs. But when it comes to PaaS, it is the vendors responsibility to manage the hardware resources [8].

However, our research on PaaS auto-scaling solutions suggests that most of the current auto-scaling solutions provided by the PaaS systems remain quite primitive (as described in Section 2.2). PaaS systems also show a tendency to stick to reactive, rule-based auto-scaling. However, unlike in an IaaS, where many different types of machines are available, PaaS auto-scaling is based on limited number of resource packages. In OpenShift these are known as gears, and in Heroku as dynos.

2.2 Related Work on PaaS Auto-Scaling

2.2.1 Openshift

Openshift is an open source PaaS project backed by Red Hat. It comes in three versions, and online version as a public PaaS, an enterprise version that can be used as a private PaaS and an open source community version. Figure 2.3 shows the architecture of Openshift as seen by a developer.

Node: a physical server or VM where all gears are located. A node typically runs an operating system like Ubuntu or RHEL.

District: a logical composition of several nodes.

Gear: an isolated environment on the operating system (node). Each node has several gears, secured and separated with SELinux, CGroups, Linux quota and Linux namespaces. User applications are deployed inside these containers.

Cartridge: pre-configured software run on a gear. Apache, PHP, Ruby, Cron and Tomcat are some examples of cartridges. OpenShift provides pre-configured cartridges for many common deployment options.

Application: software a user deploys on the platform. The cartridge or cartridges provide the automation to deliver a running application, either from source code or a pre-built artifact. An application may consist of one or many gears.

Auto-scaling decisions in Openshift are based only on the number of concurrent requests to the application. OpenShift allocates 16 connections per gear, and their utilization is monitored by the load balancer, HAProxy. If HAProxy observes a sustenance of 90% of the total connection count, a new gear is added, whereas if the demand remains at 50% or below for several minutes, a gear is removed [14].

2.2.2 Bluemix PaaS

Bluemix is a PaaS offered by IBM. It enables organizations and developers to quickly and easily create, deploy, and manage applications on the cloud. Bluemix is an implementation of IBMs Open Cloud Architecture based on CloudFoundry, an open source PaaS. Bluemix includes a reactive, rule-based auto-scaling solution. Auto-scaling rules are based on the following metrics:

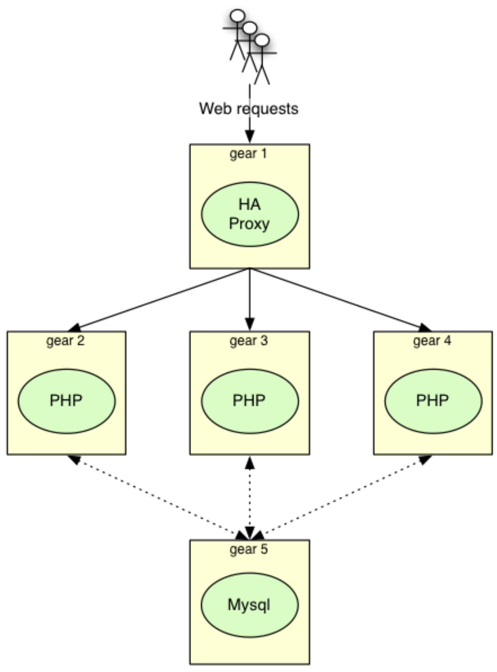


Figure 2.3: OpenShift deployment architecture

Memory: percentage usage of the memory

JVM heap: percentage usage of the JVM heap memory

Throughput: number of requests processed per second

Response time: response time of the processed requests

This multi-factored approach allow more detailed decision making. But all metrics other than memory are considered only in Java applications. Following policy fields should be provided to set rules [15]:

Allowable maximum instance count: the maximum number of application instances that can be started. Once the number of instances reaches this value, the auto-scaling service does not scale-out the application any more.

Default minimum instance count: the minimum number of application instances that can be started. If the number of instances equals this value, the auto-scaling service does not scale-in the application any more.

Metric type: the supported metric types that can be monitored.

Scale out: specifies the threshold that triggers a scaling-out action and how many instances are increased when the scale-out action is triggered.

Scale in: specifies a threshold that triggers a scaling-in action and how many instances are decreased when the scale-in action is triggered.

Statistics window: the length of the past period during which received metric values are recognized as valid. Metric values are valid only if their timestamps fall within this period.

Breach duration: the length of the past period during which a scaling action might be triggered. A scaling action is triggered when collected metric values are either above the upper threshold, or below the lower threshold, for longer than the time specified.

Cool down period for scaling in: After a scaling-in action occurs, other scaling requests are ignored during the length of the period that is specified by this parameter, in seconds.

Cool down period for scaling out: counterpart of the above parameter for scaling-out operations.

Deviating from the trend to focus on horizontal scaling, Bluemix also offers manual vertical scaling.

2.2.3 AppScale

AppScale is a PaaS solution developed with focus on building an open source version of Google App Engine (GAE) PaaS. This would facilitate distributed execution of GAE applications over virtualized cluster resources, including IaaS clouds such as Amazons AWS/EC2 and Eucalyptus [8]. The AppScale auto-scaling solution is made of the following components:

Load balancer: Nginx for static file serving and SSL, and HAProxy for health checks

App Server: Application servers running GAE apps

Database: Cassandra and ZooKeeper

Upscaling

AppScale considers Nginx itself as the head node, and it proxies traffic to HAProxy on the head node as well. Every 20 seconds, each slave node queries HAProxy to see how much traffic is coming in on each app. If more than 5 requests are queued up, waiting to be served, it will add an AppServer on that node. Maximum number of Appservers added to a node is 10. When all existing nodes are full either by appserver limit 10, reached max CPU or memory a new node is added.

Downscaling

Downscaling happens in a similar fashion. Here, if the head node notices that no requests are enqueued with HAProxy, it removes an AppServer off the first slave node, unless it is the last AppServer. Next, it moves on to the next node to do the same. If all nodes are down to their last AppServer, then a node is removed.

Although the current auto-scaling solution in AppScale remains primitive, some research has been conducted to build a pluggable auto-scaler for PaaS using AppScale [8]. It redefines high availability (HA) as the dynamic use of VMs to keep services available to users, making it a subset of elasticity (the dynamic use of VMs) which makes it possible to investigate autoscales that simultaneously address HA and elasticity.

2.2.4 Apache Stratos

Apache Stratos is an open source PaaS framework offering multi-tenancy and multi-cloud deployment capabilities. Stratos can be used to configure and deploy a PaaS on existing IaaS resources, which allows a customer to utilize existing IaaS resources by encapsulating them to the level of reduced granularity and complexity of a PaaS platform. IaaS providers compatible with Stratos include AWS , OpenStack, and GCE. Microsoft Azure support is currently under development.

Auto-scaling policy: a policy that determines the auto-scaling process, based on the load thresholds. The load thresholds in turn are determined based on the requests in flight, memory consumption and load average.

Cartridge: a container for an application, framework or data management system that could be deployed in a platform as a service (PaaS) for scalability. In Stratos, service runtimes are creating by cartridge runtimes.

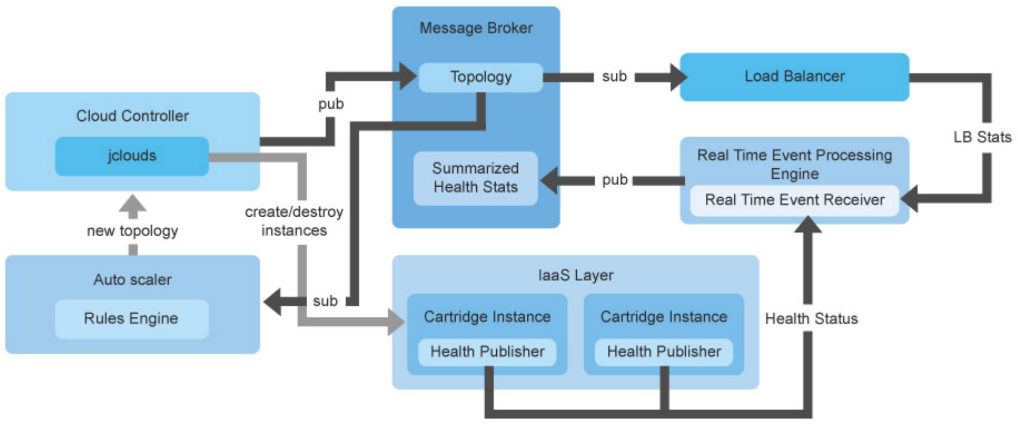


Figure 2.4: Stratos messaging architecture

Cartridge Agent: a component that resides within a cartridge instance and handles the communication between the cartridge and Stratos.

Network partition: a network bounded area in a IaaS, where private IPs can be used for communication. It is also referred to as a partition group.

Scaling down: refers to the auto-scaling system automatically shutting down the additional service instances during a non-peak-time.

Scaling up: refers to the auto-scaling system automatically spawning up the service instances during the rush time.

Architecture

Apache Stratos consists of a modular architecture with modules having different roles that are exposed as web services. Considering the auto-scaling aspect, the dominant modules are Message Broker, Auto-scaler, Cloud Controller, Load Balancer and Cartridge Agents.

Message Broker is the messaging mediator of the whole infrastructure as shown in Figure 2.4 [2]. It supports the publisher-subscriber based communication model of Stratos by allowing modules to subscribe to specific message topics (categories) and to publish messages under specific topics. The module has a Java Messaging Service (JMS) backed implementation and currently uses Apache ActiveMQ as the JMS provider.

Cloud Controller is the main control module of the system. Hence, it is the component that finally enacts the auto-scaling decisions as well. Auto-scaling operations are submitted to the underlying IaaS via the jclouds messaging API.

Load Balancer is the public facing endpoint of the application being hosted within Stratos. All requests targeted to a particular application would first arrive at the load balancer, which would then direct them to an appropriate computing instance based on a load-balancing algorithm (such as round-robin). Load balancer also keeps track of the number of pending (accepted, yet unserved) requests (essentially the request queue length or the requests-in-flight (RIF) value), which is used as an input parameter in scaling decision making.

The notion of a cartridge in Apache Stratos corresponds to an application container, similar to the Java virtual machine (JVM) in some respects. A cartridge houses the different components of the hosted application by providing the underlying technologies required by each component. For example, a traditional PHP-MySQL web application would be deployed on two cartridges; a MySQL cartridge that provides database hosting facilities, and a PHP cartridge that includes a web server (usually the Apache HTTP server) and a PHP interpreter. From the IaaS perspective, a cartridge corresponds to a single VM instance which contains the software components required by the application components intended to be deployed on it.

This cartridge-based hosting design allows Stratos to monitor the resource usage on each instance via special cartridge agents installed on each cartridge instance. Cartridge agents report health statistics (CPU load and percentage memory utilization) of the enclosing instance to the main Stratos controller node periodically (default is once every 15 seconds). Similarly, the main load balancer instance of the deployment reports the in-flight request count (for the overall system) to the controller node. These statistics are then processed using a Complex Event Processor (CEP) instance, and summarized statistics are generated to represent the latest variation of corresponding resource utilizations on a per-cluster basis (as explained in a later section).

Auto-scaler is the module responsible for analyzing the summarized statistics from each cluster and deciding the type and magnitude of scaling operation required, if any. These decisions are then transferred to the cloud controller via web service calls. Communication between different components in the above description is shown in figure 2.5[2].

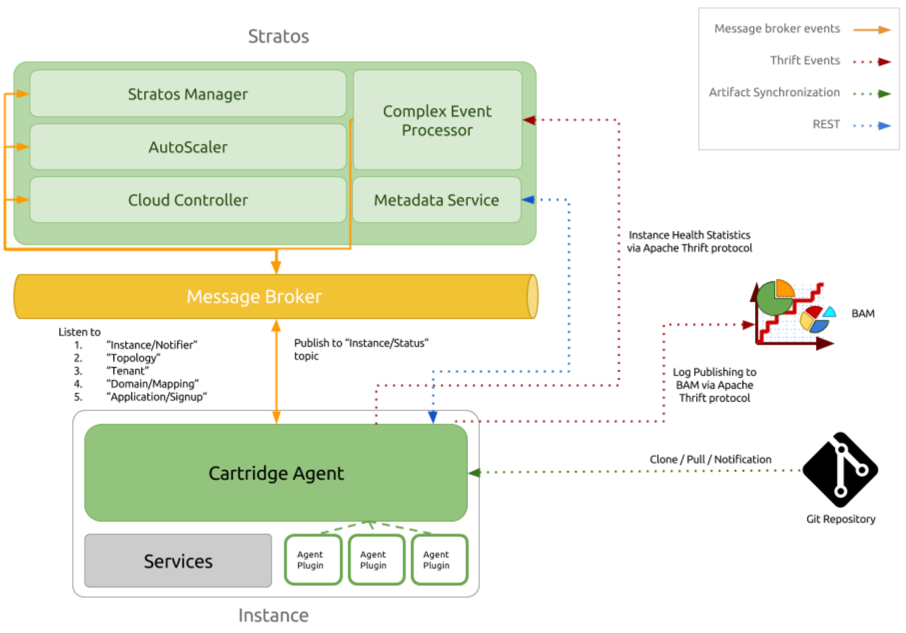


Figure 2.5: Communication diagram of Apache Stratos

Auto-scaling Policies

Each application deployed in Stratos has an associated auto-scaling policy, where thresholds are defined for each resource being considered for scaling operations (namely CPU load, memory utilization and in-flight request count). These thresholds are used as preferred resource utilization limits of the computing cluster, and scaling operations are triggered based on whether the immediate predicted load would exceed or fall below the limits.

It should be noted that these thresholds are not strict checkpoints used for making scaling decisions, in the sense that a scale-up must occur if average CPU load exceeds the threshold; rather, they describe the safe load-bearing capacity of the system, with which the required instance count of the cluster is regulated against the actual load (as described later under the auto-scaling algorithm in Section 2.2.3.5).

2.2.5 Prediction Algorithm

Both the CPU load and percentage memory consumption of each member are reported to the CEP by the cartridge agents on the running cartridge instances.

These values correspond to the overall resource usage of the computing instance (i.e., VM). However, as all processes running on the instance are uniformly affected by the effects of varying memory consumption or CPU load, and a new instance would be required to handle the load of the hosted application even if resource usage reaches a critical state due to some other process on the instance, this can be considered as a reasonable approach for acquiring resource usage statistics. The CEP also receives the current in-flight request count for each cluster, from the load balancer. The received data is then used for calculating the following statistics:

- Average requests in flight, which is the average of the number of requests in flight over a minute
- Average memory consumption per cluster
- Average CPU load per cluster (load average)

These values are then used to find averages, gradients (first derivatives) and second derivatives of the resource metrics on a per-cluster basis. Metrics are accumulated over periods of one-minute intervals called windows, and separate CEP window processors are dedicated for derivative calculations. Derivatives are calculated for all three metrics whereas averages are calculated only for the CPU and memory statistics.

Each window processor receives events continuously. When a full window of events is available, the following logic is applied to calculate gradient values:

$$\text{gradient} = \frac{v_2 - v_1}{\delta} \tag{2.1}$$

Where timestamps and values of the first and last events in the window are taken as t_1, v_1, t_2 and v_2 respectively. If the time gap between t_1 and t_2 ($\delta = t_2 - t_1$) is less than one second, it is assumed to be one second.
When calculating second derivative of a window, two gradients (first derivatives) are initially calculated: one from the first entry to the middle entry of the window, and another from the next-to-middle entry to the last entry of the window. The second derivative is taken as the gradient between these two values.

Upon calculation, the new average and gradient values are notified to the auto-scaler via an event-based mechanism. Upon receiving the events, the auto-scaler schedules a short-term prediction (extrapolation) and re-evaluation of scaling requirements.

Based on the current resource value (or average value, in case of CPU or memory usage) and the first and second derivatives, a predicted variation of the metric is calculated for the next minute using the equation:

$$s = ut + \frac{1}{2}ut^2 \tag{2.2}$$

where s is the predicted variation in the metric (for the next minute), u is the gradient of the metric, a is second derivative of the metric and t is the timespan (1 minute).

Scaling Algorithm

Predicted values (sum of current averages and predicted variations) are made available to the auto-scaling decision making logic. User-defined threshold values for each metric and the minimum and maximum numbers of instances for the computing cluster are also extracted from the system configuration as additional parameters.

Using the predicted values, the number of computing instances required for handling each of the predicted loads are determined separately. Three different equations are used for the three metrics.

In-flight request count:

$$\text{number of instances required} = ceil(\frac{\text{predicted value}}{\text{threshold}}) \tag{2.3}$$

Memory consumption: If predicted value is lower than the memory threshold, a default value (shown in equation 2.4) is returned to attempt to trigger a scale down.

$$\text{number of instances required} = (\text{minimum instances} - 1) \tag{2.4}$$

Otherwise, the following formula is used:

$$\text{required additional instances} = \frac{\text{predicted value} - \text{threshold}}{100 - \text{threshold}} \times (\text{max} - \text{min})$$

$$\text{number of instances required} = ceil(\text{minimum instances} + \text{required additional instances}) \tag{2.5}$$

This essentially distributes the fractional extra memory requirement over the full range of instances that can exist in the cluster. As an example, for a system with a 70% memory threshold and minimum and maximum cluster sizes of four and seven, a predicted load of 85% would reveal that six instances are sufficient to handle the future load.

CPU Load average:

$$\text{number of instances required} = ceil(\frac{\text{minimum instances} \times \text{predicted value}}{\text{threshold}}) \tag{2.6}$$

This essentially finds the total load of a system running the minimum instance count at the predicted load, and scales the number of instances such that the load average reaches the threshold. Unlike in the memory usage based calculation, the returned value is used even if the predicted value is below the threshold. For example, the system described above would require five instances for handling an average load of 80%. However, a predicted load of 40% would lead to an instance count of three, less than the minimum instance count, in which case the value would still be returned (and the scale down mechanism would round the value up back to the minimum).

The overall instance requirement is decided by taking the maximum of the counts calculated above (using equations (2.3) to (2.5)), and compared with the current cluster size (non-terminated instance count). If an increase is required, a scale-up is done by issuing a call to the cloud controller, subjected to the maximum instance limit for the cluster.

Unlike a scale-up, a scale-down is not effected immediately, but only when two consecutive scale-down requests are generated. This is achieved via a scale-down request counter which is reset whenever an actual scale-up or scale-down occurs in the system.

When a scale-down is performed, the full instance range is scanned to select the instance with lowest overall load, calculated by taking the mean of predicted percentages for CPU and memory consumption, which is then added to a termination-pending instance list as a new candidate for scale-down. This list is processed periodically by the cloud controller which performs the actual scale-down (member termination) operation via jclouds APIs.

Limitations and Issues

Although the auto-scaling approach described above has been in use in Apache Stratos over a considerable time (with occasional minor modifications), several limitations can be identified considering its accuracy and efficiency.

Small prediction timespan: Current algorithm only predicts the workload anticipated for the next minute. Considering the distributed statistics collection, CEP processing, internal messaging and IaaS operation delays (which can collectively exceed one minute fairly often), this prediction window needs to be expanded in order to provide a smoother response for varying load conditions.

Limited reference to past data: Currently, the algorithm only considers the last-minute time window during the prediction process. While this is sufficient for capturing the immediate workload trend of the system, it is incapable of capturing the possible periodic or seasonal workload variations of applications over larger time periods (e.g. hourly, daily or weekly). This is a major disadvantage, as certain user-oriented applications like retail websites would have clearly visible periodicities in their workload (e.g. increasing workload from morning to noon, decreasing workload at evening and minimal workload at night), which can be used to predict the general workload variation trend very accurately. Moreover, such a short time span could make the system too sensitive to minor variations in workload. This could even lead to thrashing due to rapid scale in and out decisions.

Reliance on user-defined threshold values: Currently, the user is expected to provide the threshold values for each resource as part of the auto-scaling policy of the application, which then remain constant throughout the application lifecycle. However, as the thresholds are directly related to the nature and general workload of the application, it would be difficult for an inexperienced user to plug in correct values for the thresholds. Although it is possible to dynamically adjust the threshold values based on past workload data, such a strategy has not yet been implemented in the Stratos autoscaler.

Homogeneous instance spin-up: All commercial IaaS providers now offer a wide range of VMs, many of them with specialized resource capabilities. For example, Amazon EC2 offers different classes of instances such as compute (CPU) optimized, memory optimized and storage optimized. Spinning up such specialized instances would be beneficial in most cases, since a cluster would likely run out of only one of the resource types at a given time. Also, it may not be possible to spin up a large number of nodes of the same resource type, as some license policies (e.g., Client Access Licenses (CAL) for Windows-based services) allow only a limited number of clients, regardless of the performance of a single client. However, current resource management policy of Stratos is inherently homogeneous, where the same predefined type of instance is spinned up in all scale-up operations.

Insensitivity to the IaaS cost model: Different IaaS providers maintain different costing policies for their VM instances and other computing resources. For example, Amazon EC2 maintains a hourly charging policy for VMs, where a running VM is charged at the start of each hour of its uptime. This exposes the possibility of "smart killing" where an instance selected for spin-down can still be kept running until the current (already-paid) hour elapses, considering the possibility that it might be useful in handling an unexpected spike within that hour, rather than having to spin up a new instance to handle it on time. Other vendors like Google Compute Engine maintain different pricing policies, and as a result the current implementation does not take into consideration any of these policies during scaling operations.

Usage of percentage values in place of absolute workload statistics: Currently, memory usage statistics are converted to percentages by cartridge agents before being dispatched to the CEP. While this is currently acceptable as the system always spins up instances of the same type (having identical CPU and memory characteristics), it would be problematic if heterogeneous instance spawning has to be implemented, as it would not be possible to determine the absolute resource utilization of the system from percentages alone.

Considering the above facts, we can identify two main opportunities for improvement in the Stratos autoscaler:

- a better workload prediction strategy
- a resource requirement and cost-aware resource (instance) allocation strategy

2.3 Workload Prediction

As described in Section 2.1, the lack of anticipation of a reactive approach clearly affects the overall auto-scaling performance due to two reasons [4][6]:

- Reactive systems might not be able to effectively handle sudden traffic bursts or continuously varying traffic patterns.
- The time it takes to process the auto-scaling actions affects user goals like cost and SLA. For example, if the VM instantiation time is about 5 minutes in a scale-up action, SLA violations might occur over that period of time. Alternatively, for IaaS providers who provide hourly based pricing scheme, if a scale-down action is requested in the last few minutes of the current working hour and that request gets completed only in the next hour, the user would have unintentionally paid for next hour as well, although he/she does not utilize the VM in the next hour.

A desirable solution would require an ability to predict the incoming workload on the system and take auto-scaling actions in advance. This capability in turn will enable the application to be ready to handle load increments when they actually occur.

2.3.1 Workload and Resource Usage Patterns

When provisioning resources for dynamically changing workloads and resource usage patterns, it is important to identify and classify the patterns involved. According to Mao and Humphrey

[16] there are four representative workload pattern in the cloud environment as shown in Figure 2.6 These pattern explain the typical application behaviors in cloud computing.

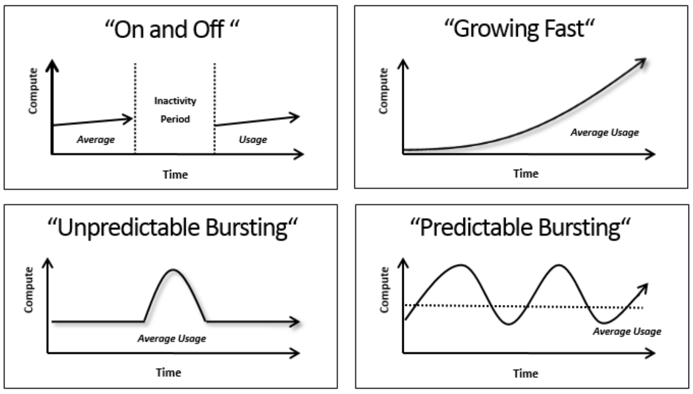


Figure 2.6: Categories of cloud workload patterns

Growing workload: This pattern may represent a scenario in which some piece or content of the application suddenly becomes popular. From resource usage point of view, this pattern represents loading of large data sets to memory.

Predictable bursting/cyclic workload: This pattern represent services with seasonality trends; for example an online retailer, in which case daytime has more workload than the night, and holiday and shopping seasons might anticipate more traffic than normal.

On-and-off workload pattern: This represents work to be processed periodically or occasionally, such as batch processing jobs and data analysis performed daily.

Unpredictable bursting: Unexpected peaks in demand occur in this type of pattern. A typical scenario would be spikes occurring in the CPU usage, or in the incoming request count. Since this type of bursting usually does not last long, over-provisioning is not a suitable approach for this type of pattern.

In addition, stable workloads may also be observed in many cases, which are characterized by a constant number of requests, or CPU or memory usage, per minute. For this type of workload pattern, forecasting is not very challenging.

This categorization shows the abstractions of some major workload patterns. But real-world workload and resource usage patterns are more complex than this. Typical applications have combinations of above patterns [4]. An example real world workload is shown in 2.7

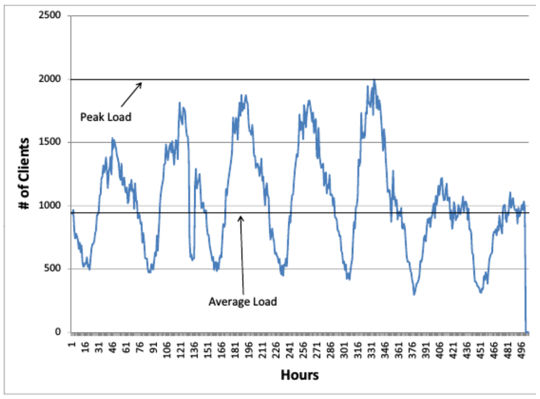


Figure 2.7: Workload for 1998 Soccer World Cup website

2.3.2 Prediction Techniques

Proactive auto-scaling has two major concerns:

1. Prediction of a future value of a certain performance metric based on past values
2. Arriving at a scaling decision based on the predicted value

This section reviews the forecasting techniques which can be used for the first purpose. These techniques can be used to forecast the performance metrics like workload metrics (request count) or resource usage metrics (CPU, memory, or I/O usage). There are two main approaches for predicting workload and resource usage [6]:

1. Time series analysis techniques
2. Machine learning techniques

Time series analysis

A Time series is a sequence of measurements of some performance matrix, typically measured at successive time instants spaced at uniform time intervals. Goal of time series analysis are the forecasting of a future value and identifying repeated patterns in the series.

Input for a time series algorithm is a sequence of the last w observations. The size of the input is known as the input window (w).

$$X = x_t, x_{t-1}, x_{t-2}, ..., x_{t-w+1}$$

Time series components
In time series analysis, the time series is decomposed into basic components and separate analysis techniques are used to identify each component [17]. A detailed decomposition of workload data is shown in 2.8[18]

Trend: A trend exists when there is a long-term increase or decrease in the data. It does not have to be linear.

Seasonal pattern: A seasonal pattern exists when a series is influenced by seasonal factors (e.g., the quarter of the year, the month, or day of the week). Seasonality is always of a fixed and known period.

Cyclic pattern: A cyclic pattern exists when data exhibits rises and falls that are not of a fixed period.

X_t, the data at period t, can be expressed as sum of St, the seasonal component at period t, T_t, the trend-cycle component at period t, and E_t, the remainder:

$$X_t = S_t + T_t + E_t$$

Time Series Predictions The widely used techniques in the literature for prediction purposes are smoothing techniques like moving average, exponential moving average, autoregressive model, and ARMA (autoregressive moving average).

Averaging Methods These methods are typically used to smooth a time series in order to remove noise or to make predictions. Smoothing always involves some form of local averaging of data such that the nonsystematic components of individual observations cancel each other out. Smoothing is usually done to help us better see trends component in the time series by

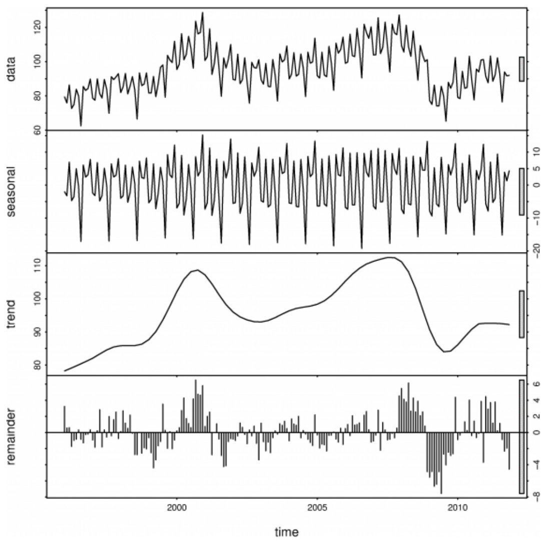


Figure 2.8: Components of a time series

smoothing out the irregular roughness to see a clearer signal. Smoothing does not provide us with a model, but it can be a good first step in describing various components of the series [6][17].

In general, a forecast value Y_{t+1} can be calculated as a weighted sum of last w consecutive values:

$$Y_{t+1} = a_1 x_t + a_2 x_{t-1} + a_3 x_{t-2} + ... + a_w x_{t-w+1} \; where \; a \geq 0 \; \sum_{i=1}^{w} a_i = 1 \tag{2.7}$$

By introducing some constraints on weights, we can arrive at several derivations of the averaging method.

Simple moving average, SMA(q): Simple MA is the arithmetic mean of the last q of w values, so that the weights of all the observations are the same:

$$Y_{t+1} = \frac{x_t + x_{t-1} + x_{t-2} + ... + x_{t-q+1}}{q} \; so \; that \; a_i = \frac{1}{q} \; for \; 1 \leq i \leq q \tag{2.8}$$

Weighted moving average, WMA(q): Weights associated with each observation are different. Typically more weight is given to the most recent terms in the time series.

$$Y_{t+1} = \frac{a_1 x_t + a_2 x_{t-1} + a_3 x_{t-2} + ... + a_q x_{t-q+1}}{a_1 + a_2 + a_3 + ... + a_q} \tag{2.9}$$

Exponential weighted moving average: In exponential weighted moving average methods, forecasts are calculated using weighted averages where the weights decrease exponentially as observations come from further in the past. The smallest weights are associated with the oldest observations [17][24][25].

Simple exponential smoothing:

$$Y_{t+1} = \alpha x_t + (1 - \alpha) Y_t \tag{2.10}$$

$$Y_{t+1} = \alpha x_t + \alpha(1 - \alpha) x_{t-1} + \alpha(1 - \alpha)^2 + ... + \alpha(1 - \alpha)^{w-1} x_{t-w+1} \tag{2.11}$$

where 0 1 is the smoothing parameter. The one-step-ahead forecast for time T+1 is a weighted average of all the observations in the series x1, , xT. The rate at which the weights decrease is controlled by the parameter . If is small (i.e. close to 0), more weight is given to observations from the distant past. If is large (i.e., close to 1), more weight is given to the more recent observations. This method is suitable for forecasting data with no trend or seasonal patterns.

Double exponential smoothing: Double smoothing can be applied to a time-series with an existing linear trend. This can be derived by applying exponential smoothing to the data already smoothed using simple exponential smoothing. There is a need to add a second smoothing constant to account for trend.

Triple exponential smoothing: Triple smoothing can be applied to a time-series with existing trend and seasonality. This can be derived by applying exponential smoothing to the data already smoothed using double exponential smoothing.

ARMA (p, q) (Auto regressive Moving Average) Models

This model combine two models called auto regression (of order p) and moving average (of order q).

Auto regression model (p): In an auto regression model, the variable of interest is forecasted using a linear combination of past values of the variable. The term auto regression indicates that it is a regression of the variable against itself. Thus an auto regressive model of order p can be written as

$$X_t = c + \Phi_1 x_{t-1} + \Phi_2 x_{t-2} + ... + \Phi_p x_{t-p} + e_t \tag{2.12}$$

where c is a constant and e_t is white noise. Auto regressive models are normally restricted to stationary data, and then some constraints on the values of the parameters are required [17].

- For an AR(1) model: $1 < \Phi_1 < 1$
- For an AR(2) model: $1 < \Phi_2 < 1, \Phi_1 + \Phi_2 < 1, \Phi_2\Phi_1 < 1$

Moving average model MA(q): Moving average model uses past forecast errors in a regression-like model.

$$X_t = c + e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_q e_{t-q} \tag{2.13}$$

where et is white noise. Xt can be thought of as a weighted moving average of the past few forecast errors. However, moving average models should not be confused with moving average smoothing. The former is used for forecasting future values while the latter is used for estimating the trend-cycle of past values.

If we combine differencing with auto regression and a moving average model, we obtain a non-seasonal ARIMA model. ARIMA is an acronym for Auto Regressive Integrated Moving Average (integration in this context is the reverse of differencing). The full model can be written as:

$$X_t = c + \Phi_1 x_{t-1} + \Phi_2 x_{t-2} + ... + \Phi_p x_{t-p} + e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_q e_{t-q} \tag{2.14}$$

The predictors on the right hand side include both lagged values of X_t and lagged errors. This is called this an ARIMA(p, d, q) model, where p and q are the orders of the auto regressive and moving average parts respectively, and d is the degree of first differencing involved.

Machine Learning Techniques

Regression

Regression is used to determine the polynomial function that is closest to a set of points (in this case, the w values of the history window). The objective is to find a polynomial such that the distance from each of the points to the polynomial curve is as small as possible. In other words, regression finds a polynomial that best fits the data. Linear regression is a special case, where the order of the polynomial is one. Linear regression is used heavily in literature, with the input window of the time series as its input data set.

Neural Networks

This is the most widely used machine learning technique in the literature. A neural network can be thought of as a network of artificial neurons organised in layers. The predictors (or inputs) form the input layer, and the forecasts (or outputs) form the top layer [6][17]. There may be intermediate layers containing hidden neurons, and one neuron for the predicted value in the output layer. During the training phase, it is fed with input vectors and random weights. Those weights will be adapted until the given input shows the desired output, at a learning rate ρ.

Review on Workload and Resource Usage Prediction Techniques

Among the two main streams of workload predictions techniques, time series techniques have been applied mostly in the literature in the context of workload and resource usage prediction.

Simple moving average is rarely used for prediction purposes due to poor performance, but it is a widely used technique for noise reduction [19]. Specially, moving average technique provides acceptable results for workloads with lots of spikes, because it will not be cost effective to scale according to such highly varying workloads.

Exponential smoothing is popularly used for predicting purposes. Mi et al. [20] also used a quadratic exponential smoothing against real workload traces (World Cup 98), and showed good accurate results, with a small amount of error. Some researches show that exponential smoothing performs better than both moving average and weighted moving average methods.

The auto-regression method has been widely used in the literature ([21], [22], [23]). Kupferman et al. [21] applied autoregression of single order to predict request rate and found that its performance depends largely on several user-defined parameters: the size of the input window and the size of the adaptation window.

Auto-regressive moving average method (ARMA) is the dominant time series analysis technique for workload and resource usage prediction. Roy et al. [4] use a second order ARMA for workload prediction, based on the last three observations. The predicted value is then used to estimate the response time calculation for scaling decisions.

Machine learning techniques like neural networks and regression have been applied by several authors but those techniques are not widely used in the literature. The reason is that such techniques incorporate significant model creation (training) time before actual usage. On the other hand, these systems have performed less accurately during initial stages of the predictions, but eventually they tend to perform accurately. As a result, implementations like neural networks have been trained offline while time series techniques were being used to do the actual predictions, until the the neural networks produced significantly accurate results. Nevertheless, there were researches on using history window values as the input for a neural network [24] and a multiple linear regression equation [21], [24]. The accuracy of both methods depends on the input window size.

Another interesting point in workload prediction is estimating the prediction interval, r. Islam et al. [24] proposed using a 12-minute interval, because the set-up time of VM instances in the cloud was typically around 5-15 minutes by that time. Time-series forecasting can be combined with reactive techniques. Iqbal et al. [25] proposed a hybrid scaling technique that utilizes reactive rules for scaling up (based on CPU usage) and a regression-based approach for scaling down.

There is a significant amount of already established research work in the workload prediction domain. But most of the research work in cloud workload prediction shows that some techniques work only if certain constraints are satisfied. Constraints could be the applicability to a particular workload pattern, or some kind of optimum parameters adjustment. Several prediction techniques will not work with some workload patterns while several other techniques might work well under the same conditions. Hence, parameter adjustment and applicability for unseen workload pattern are some of the challenges in predicting the workload and resource usage in cloud infrastructure.

2.4 Resource Allocation

2.4.1 Quality of Service

When a SaaS solution is provided over the cloud, the SaaS consumer or the end user of the software expects some kind of guarantee from the SaaS provider. To guarantee quality of service (QoS) for customer satisfaction, therefore, a Service Level Agreement (SLA) is implemented between customers and SaaS providers [26]. But most of the time, these SaaS providers rely on a PaaS to deploy their services. Therefore, PaaS providers are also expected to guarantee QoS to PaaS consumers through a SLA.

However, deciding a QoS required by a particular PaaS consumer is complicated by multiple factors:

1. Higher QoS requires higher availability which results in higher costs.
2. Lower QoS might result in losing existing customers and possibly new customers.
3. QoS requirement of PaaS consumer may change from time to time.
4. There is no standard over the metrics considered in deciding QoS.

Considering these factors, we argue that SLA should be a major factor in auto-scaling process.

Related Work

The SLA-based provisioning model proposed in [26] (2.9) can be adapted for PaaS-based auto-scaling solutions as well. The solution is aimed at maximizing customer satisfaction level and minimizing cost. It is proposed on the basis of customer-driven resource management rather than a pure cost minimizing approach. Key considerations in the proposed model are:

1. Customer profiles such as credit levels (here customer refers to the to SaaS user)

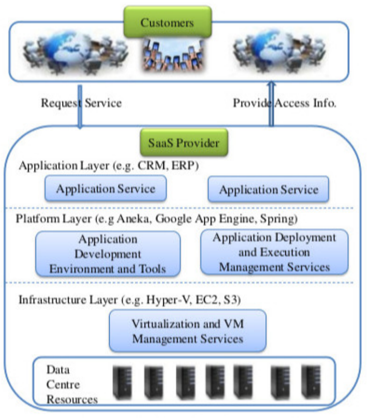

Customers
Request Service
Provide Access Info.
SaaS Provider
Application Layer (e.g. CRM, ERP)
Application Service
Application Service
Platform Layer (e.g Aneka, Google App Engine, Spring)
Application Development Environment and Tools
Application Deployment and Execution Management Services
Infrastructure Layer (e.g. Hyper-V, EC2, S3)
Virtualization and VM Management Services
Data Centre Resources

Figure 2.9: Layered view of the SLA-based provisioning model

2. Key performance indicators
   Key performance indicators can be evaluated based on financial agility, assurance, accountability, security and privacy, usability, and performance [27]. Here, the model has considered:

   from the provider's perspective: cost (under financial category): the total cost of resource usage including VM and penalty cost

   from the SaaS users perspective: service response time (performance) & SLA violations (assurance): The possibility of SLA violations creates a risk for SaaS providers. SLA violations are caused by elapse in the expected response time, and whenever a SLA violation occurs, a penalty is charged.

Challenges faced are:

- How to manage dynamic customer demands (such as upgrading from a standard product edition to an advanced product edition, or adding more accounts)
- How to reserve resources, considering customer profiles and multiple KPI criteria
- How to map customer requirements to infrastructure-level parameters
- How to deal with infrastructure level heterogeneity (such as different VM types and service initiation times)

Actors and Roles of the model

SaaS Provider: SaaS providers lease web-based enterprise software as services to customers. The main objective of SaaS providers is to minimize cost and SLA violations. The SaaS provider will act as a PaaS consumer if the SaaS is deployed using a PaaS system rather than directly on an IaaS. Here we consider that the SaaS provider provides a product with multiple editions (e.g., Standard, Professional, Enterprise). Also, these editions comes with SLAs with different QoS guarantees.

Customer: A company or a person registered in the SaaS providers portal, providing information required to consume an edition of the product, is considered a customer. For the model customer-related information such as the number of employees to be serviced and type of business enterprise in terms of capacity (small, medium, large) is considered.

Although the model is mostly focused on SaaS systems, we believe understanding the requirements of SaaS providers (PaaS consumers) would be beneficial for providing an application-aware PaaS auto-scaling solution. In the SaaS providers point of view, this model would require

PaaS layer to include mapping and scheduling policies to reflect SaaS providers requirements at IaaS level .

2.4.2 Costing

Cost plays a major role in all on-demand resource allocations. Hence, it is a major concern of all auto-scaling approaches. For example, Alipour et al.[7] have identified the following factors as essential components of a cloud auto-scaler:

Responsiveness of the entity performing scaling operations: lack of responsiveness leads to delayed resource allocation, often leading to unnecessary costs since it may already be too late to fulfill the earlier requirement.

Cost effectiveness: lack of cost awareness can lead to allocation of resource units that are arbitrarily large or small relative to the actual requirement, leading to over-provisioning or under-provisioning that ultimately results in excessive cost or expensive SLA violations, respectively.

Appropriate handling of the failure of scaling operations: if the system does not have a feedback mechanism to handle scaling operation failures, inability to cope with failed scale-down operations would lead to retainment of extra resources, resulting in unnecessary costs; on the other hand, inability to handle failures of scale-up operations would result in insufficient resources and failure to meet service level objectives (SLOs).

We can view the pricing concern from different perspectives:

- For a PaaS provider (who leases resources as an IaaS customer and provides it to his own clients), the goal would be minimizing the allocation of IaaS resources and distributing allocated resources over a maximum number of PaaS customers to maximize his revenue.
- For a PaaS user, the goal would be to minimize the resource allocation cost for his own application while meeting the required QoS levels.

If we leave multi tenancy out of our scope, both problems get simplified into the single problem of minimizing the cost of IaaS resource allocation while meeting the QoS expectations of the PaaS user.

A major challenge in consolidating resource prices into a generic model is the wide variation of resource types and capacities across various IaaS providers (for example, AWS offers a largely different array of VM specifications than GCE), and the non-uniformity of their pricing policies (for example, the hourly billing scheme of Amazon EC2 against the more complicated billing scheme of GCE, where instances are initially billed for the first 10 minutes of operation and then incrementally billed for each minute of additional usage). Some vendors also show a rapid evolution of their pricing policies over time (for example, Amazon EC2 alone has changed its prices 44 times from 2006 to early 2015 [28]).

Techniques

Reinforcement learning (RL) is a principal technique used in most cloud auto-scalers [6]. This is essentially a feedback mechanism via which the auto-scaler is made aware of the effects of its scaling decisions. Under RL, the auto-scaler receives representation of the system state at a given time (in terms of workload, performance or other variables), along with an associated reward value (a heuristic representation of the effectiveness of the said state). The goal of the auto-scaler is the maximization of cumulative future rewards by varying the allocated resources. Since actions like scaling up may lead to infinite rewards (as the QoS parameters of the system continue to increase with resource allocation), a discount rate is introduced to hinder the continued increase of reward with allocation. Dynamic programming, Monte Carlo methods and temporal-difference learning are some of the main techniques used in reinforcement learning. Unfortunately, this technique alone cannot be used to make reliable resource allocation decisions, since it shows bad initial performance and requires a considerable training time to reach sufficient maturity. Curse of dimensionality is another issue, implied by the large state space that has to be dealt with by the policy mapping. For the latter issue, certain nonlinear function approximators including neural networks, regression trees and support vector machines have been proposed as mapping functions, instead of nave lookup tables.

Look-ahead optimization, utilizing the principle of receding horizon control, is another approach used for solving the resource optimization problem over a predefined time horizon (up to some N units of time), considering current and future constraints into account [4]. It is essentially a state-space search, which results in the construction of a state transition tree with a branching factor equal to the number of finite input choices, and depth (N + 1). The optimal resource allocation scheme is decided by tracing a path from the root (current state) to the lead node in the search tree, such that the leaf is closest to the intended destination state and the net cost across all branches is minimized.

Related Work

IaaS level

Due to the relative novelty of PaaS provisioning with respect to IaaS, existing research mostly focuses on cost modelling at the IaaS level. Fortunately, many of them can be adapted to PaaS with minor modifications.

Kupferman et al. [21] initially presented with the idea of smart killing, which has been described earlier. Since most of the existing IaaS providers allocate resources on a pay-in-advance basis, smart killing has become an integral part of many auto-scaling cost models. The research also proposes a scoring algorithm for comparing heterogeneous auto-scaling techniques, although the cost model considered for scoring is quite limited in the sense that it does not take different VM types and certain other cost elements into account, and requires manual tuning of several parameters through experimentation.

One of the earliest proposals for heterogeneous resource allocation is presented in [29]. It brings up the following concerns to justify the selection of cost-effective instance types for auto-scaling:

- Availability of virtually unlimited resources, under a limited budget
- Non-ignorable acquisition time for VM instances
- Pay-in-advance billing model s (e.g. full-hour model on Amazon EC2)
- Availability of multiple instance types

The authors proposed a four-component system model (performance monitor, history repository, auto-scaling decider and VM manager). Evaluation of the model has demonstrated a cost saving of 20% to 45%, compared to the use of fixed instance types.

A performance, resource and cost aware resource provisioning approach for the IaaS is discussed in [30]. Here, a scaling decision is associated with three cost components: an actual resource (lease) cost, a reconfiguration cost (service degradation or downtime resulting from the reconfiguration), and an income measure associated with guaranteeing a particular QoS. Collectively, these metrics evaluate the future benefits of a configuration change against the actual reconfiguration cost. A number of situation and IaaS-specific parameters are required for producing an accurate decision based on this approach.

A similar proposal of deciding on reconfiguration over brand new resource allocation has been brought up in [31]. The approach, called Smart Cloud Optimization for Resource Configuration Handling (SCORCH), is a model-driven engineering (MDE) approach. It uses a queuing model to represent available resources (VMs), and works by checking the resource requirement of a requested scaling operation against the VMs standing in the queue, and picks a matching instance if one is available. If an exact match (in terms of resource capacities) cannot be found, it is attempted to find a running VM configuration which can be modified to match the requirement more quickly than provisioning a new VM. A set of resource features like CPU type, operating system, and type and version of installed application server(s) are used to model the queue of VMs. SCORCH is known to yield up to a 50% reduction of power consumption and cost.

A multi-factor pricing model including request submission time (peak vs. off-peak), nature of pricing rates (fixed vs. changing) and availability of resources (supply and demand) is proposed in [32]. Here, resource allocation is targeted at optimizing customer revenue, using a request queuing model against mean and instant response time.

A significantly different approach for cloud resource modelling is taken in [33]. Here, resources and services are defined in terms of a Web Service Modelling Ontology (WSMO), and semantic techniques are used for service discovery. It may be possible to extend this technique for resource allocation as well, since it has already been possible to formulate semantic SLAs for discovery of monitoring services.

Other proposed approaches to resource allocation in IaaS-level auto-scaling include usage of energy costs to derive an auto-scaling algorithm for dynamic and balanced provisioning of resources [34], generating a global performance-to-price model via game theory by treating each application as a selfish player competing to guarantee its QoS requirements while minimizing resource costs [35], modelling the application response times based on different resource allocation combinations [36] and coming up with optimal resource scheduling plans using deadline-assignment concepts and instance consolidation (running multiple schedules on instances subjected to a smart-killing time window) [ref [70] in [6]].

PaaS level An important contribution to cost-model-based auto-scaling in PaaS is presented in [8]. It introduces a composite auto-scaler for the AppScale PaaS, with three components providing awareness of high availability, quality of service (QoS) and cost. Targeting on Amazon EC2, the cost-aware auto-scaling component utilizes the smart killing technique to retain hot spares (i.e., extra, execution-ready instances) for handling sudden loads.

In addition, the AppScale auto-scaler purchases spot instances (low-cost, on-demand-allocated instances available for public bidding) available on EC2 as hot spares. Spot instances, while having a larger acquire time than dedicated instances, can be spawned aggressively for QoS maintenance and fail-over recovery. Evaluations have revealed that the cost-aware component produces a 91% saving over the optimizations already achieved by the high availability (HA)-aware and QoS-aware components. While this is a notable achievement, the module is too tightly coupled with Amazon EC2 to be called a generic approach for capturing IaaS cost models.

2.5 Performance Evaluation

There is no standard method of evaluating auto-scaling techniques. Researchers use various types of methods, each of which differ according to the performance metrics they use. Most of the researchers tend to build their own custom testbeds, suitable for their research objectives. However, there is a common pattern in all of them: evaluation requires a real or realistic environment, using simulators, real cloud providers or custom testbeds [6]. Regardless of the selected experimental platform, a controlled environment is required to do the experiments. Two types of workloads are used for these purposes: synthetic and empirical. Synthetic workloads are programmatically generated while empirical workloads are traces from real-world applications. Finally, an application benchmark is required to execute the simulation in real cloud providers or custom testbeds. In this section we review the available simulation platforms, frequently used workloads and application benchmarks commonly found in research.

2.5.1 Experimental Platform

Traditional and emerging cloud based application services include social networking, web hosting, content delivery and real-time data processing. They differ from each other in their configurations, deployment requirements and composition. Quantifying the performance of each of these application models in a real cloud system is extremely challenging because:

1. Clouds exhibit varying demands, supply patterns, system sizes and computing resources.
2. Cloud users have heterogeneous, dynamic and competing QoS requirements.
3. Cloud applications have varying performance, workload, and dynamic application scaling requirements.

Use of real-world systems, such as Amazon EC2 and GCE, for benchmarking application performance under varying conditions is often limited due to their cloud infrastructure. It is extremely difficult to re-configure benchmarking parameters across a massive-scale cloud computing infrastructure over multiple test runs to obtain reliable results by reproducing the same test over and over again. Cloud computing infrastructures are not under the control of the developers of application services, hence it is not possible to perform benchmarking tests in a repeatable, dependable, and scalable environment when using real-world cloud environments.

An alternative method which is viable and minimizes the experimentation cost, is the use of simulation tools. This has a cost in terms of the system configuration effort. These tools open up the possibility to test multiple hypotheses in a controlled environment where experiments can be easily repeated to obtain more reliable and accurate results. Simulation tools provide much advantage over real-world cloud systems by providing:

1. Ability to test services in a repeatable and controllable environment
2. Tune system bottlenecks before deploying on a real cloud
3. Experiment with different workloads and resource performance scenarios

Simulation platforms can be implemented either by selecting an already existing simulator or by building custom software from scratch. Most of the existing simulators have their own capabilities and limitations which might not suit the need of some application services, in which case building custom software from scratch is the only option. Some research oriented cloud simulators are CloudSim, MDCSim and GreenCloud.

CloudSim

CloudSim is a new, generalized, and extensible simulation framework that allows seamless modeling, simulation, and experimentation of emerging Cloud computing infrastructures and application services. Using CloudSim, developers can test the performance of a newly developed application service in a controlled environment. CloudSim offers the following novel features [37]

- Support for modeling and simulation of large-scale cloud environments
- A self-contained platform for modeling clouds, service brokers, and allocation and provisioning policies
- Support for simulation of network connections among the simulated elements
- Simulation of a federated cloud environment that inter-networks resources from both private and public cloud domains, a critical feature related to research on cloud-bursting and automatic application scaling
- Virtualization engine that creates and manage multiple, independent and co-hosted virtualized services
- Flexibility for switching from space-shared and time-shared allocation of resources ranging from processing cores to virtualized services

The main advantages of using CloudSim for initial performance tests include, time saving, flexibility and applicability.

MDCSim

MDCSim is a comprehensive, flexible and scalable platform to simulate a multi-tier data center. Simulation experiments are possible with large numbers of nodes across the cluster, making it scalable and apt to existing data centers. MDCSim considers the effect of all tiers and can accurately estimate the power consumption across the servers of a multi-tier data center [19].

GreenCloud

GreenCloud is developed as an extension of a packet-level network simulator NS2. Unlike CloudSim and MDCSim, GreenCloud extracts, aggregates and makes available in an unprecedented fashion, information about the energy consumed by computing and communication elements of the data center [38].

2.5.2 Experimental Workloads

To auto-scale cloud computing resources according to the fluctuating workload, it is important to understand the workload characteristics and measure the sensitivity of the application performance to different workload attributes. Synthetic workloads can be generated based on different patterns. There are four representative workload patterns in cloud environments, namely stable, growing, cycle/bursting and on-and-off (see Figure 2.4).

The commonly used technique for workload generation are user emulation and aggregate workload generation. In user emulation, each user is emulated by a separate thread that mimics the actions of a user by alternating between making requests and lying idle. The attributes for workload generation in the user emulation method include think time, request types, inter-request dependencies, etc. User emulation allows fine-grained control over modeling the behavioral aspects of the users interacting with the system under test. However, it does not allow controlling of the exact time instants at which the requests arrive the system. Aggregate workload generation is another approach that allows specifying the exact time instants at which the requests should arrive the system under test. However, there is no notion of an individual user in aggregate workload generation. Therefore, it is not possible to use this approach when dependencies between requests need to be satisfied. Dependencies can be of two types: inter-request and data dependencies. An inter-request dependency exists when the current request depends on the previous request, whereas a data dependency exists when the current requests requires input data which should be obtained from the response of the previous request [6].

Synthetic Workload

An important requirement for a synthetic workload generator is that the generated workloads should be representative of the real workloads and should preserve the important characteristics of real world workload, such as inter-session and intra-session intervals. There are two types of approaches to generate workload:

Empirical Approach: Traces of real-world applications are sampled and replayed to generate synthetic workloads. This approach lacks flexibility because it only represents the particular application, load conditions and system configurations on which the workload data was obtained from.

Analytical Approach: This is the use of mathematical models to define workload characteristics that are used by a synthetic workload generator. It is possible to modify workload model parameters one at a time and investigate the effect on application performance, and to measure application sensitivity for different workload attributes.

Many applications are being widely used in research for synthetic workload generation, which can be used to generate simple workload patterns based on different attributes. Few of the commonly used applications are listed below.

Httperf
Httperf is a tool that generates various HTTP workloads for measuring server performance. It has a core HTTP engine that handles all communications with the server, a workload generation module that is responsible for initiating appropriate HTTP calls at the appropriate times, and a statistics collection module [39].

SURGE
This is a request generation tool for testing network and server performance. It uses an offline trace generation engine to create traces of requests. Web characteristics such as file sizes, request sizes, popularity and temporal locality are modeled statistically. Pre-computed data-sets consist of the sequence of requests to be made, the number of embedded files in each web object to be requested, and the sequences of active and inactive OFF times to be inserted between requests. SURGE outputs a workload that agrees with the six distributional models that make up the SURGE model (file sizes, request sizes, popularity, embedded references, temporal locality, and OFF times) [40].

SWAT
SWAT is a tool for stress testing session-based web applications. It uses a trace generation engine that takes session lets (a sequence of request types from a real system user) as input and produces an output trace of sessions for the stress test. SWAT uses httperf for request generation [40].

Faban
This is a Markov-based workload generator which provides facilities to develop and run work-

load benchmarks. It has two major components, the Faban Driver Framework which is an API-based framework and component model to build benchmarks, and the Faban harness to automate server benchmarks [41]

Rain

Rain is a statistics-based workload generation toolkit that provides a thin, reusable, configuration and load scheduling wrapper around application-specific workload request generators, which can easily use parameterized or empirical probability distributions to mimic different classes of load variations [42].

Empirical Workload

We were unable to find any publicly available empirical workloads from cloud providers. In the literature we reviewed, most of the authors generated their own traces using private cloud systems, running benchmark applications or real-world cloud applications. However, we encountered a few workloads that have been referenced by most of the researchers as baselines for their researches.

1998 Soccer World Cup

The 1998 Soccer World Cup trace is extensively used in literature. This dataset consists of all the requests made to the 1998 Soccer World Cup web site between April 30, 1998 and July 26, 1998. During this period of time, the site has received 1,352,804,107 requests [43].

ClarkNet

This constitutes two traces containing two weeks worth of all HTTP requests to the ClarkNet WWW server. ClarkNet is a full Internet access provider for the Metro Baltimore-Washington DC area [44].

Google Cluster Data

These are two traces of workloads running on Google compute cells. ClusterData2011_2 provides data from a 12.5k-machine cell over about a month in May 2011. TraceVersion1 is an older, shorter trace that describes a 7-hour period from one cell (cluster) [45].

2.5.3 Application Benchmarks

Application benchmarks are used to evaluate server performance and scalability. Typically, they comprise a web application together with a workload generator that creates synthetic session-based requests to the application [40].

RUBiS

RUBiS is an auction site prototype modeled after eBay.com, used to evaluate application design patterns and application servers performance scalability. The benchmark implements the core functionality of an auction site: selling, browsing and bidding. No other complementary services like instant messaging or newsgroups are currently implemented. In RUBiS there are three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during visitor sessions, during a buyer session users can bid on items and consult a summary of their current bids, and ratings and comments left by other users. Seller sessions require a fee before a user is allowed to put up an item for sale. An auction starts immediately and lasts typically for no more than a week. The seller can specify a reserve (minimum) price for an item. The application consists of three main components:

Apache load balancer server, JBoss application server and MySQL database server [46].

RUBBoS
RUBBoS is a bulletin board benchmark modeled after an online news forum like Slashdot. RUBBoS implements the essential bulletin board features of the Slashdot site. In particular, as in Slashdot, it supports discussion threads. A discussion thread is a logical tree, containing a story at its root and a number of comments for that story, which may be nested. Users have two different levels of authorized access: regular user and moderator. Regular users browse and submit stories and comments. Moderators additionally review stories and rate comments [47]

TPC-W
TPC-W specifies an E-commerce workload that simulates the activities of a retail store website. Emulated users can browse and order products from the website. In the case of TPC-W, the products are books. A user is emulated via a Remote Browser Emulator, RBE, that simulates the same HTTP network traffic as would be seen by a real customer using a browser. The TPC-W primary metrics are the WIPS rating and system cost per WIPS. A WIPS is the number of Web Interactions Per Second that can be sustained by the System Under Test, SUT (the collection of servers that provide the TPC-W E-commerce solution). The cost per WIPS is essentially the cost of the SUT divided by the WIPS rate. The TPC-W benchmark applies the workload via emulated browsers [48]

CloudStone
CloudStone is a multi-platform, multi-language performance measurement tool for Web 2.0 and cloud computing. CloudStone involves using a flexible, realistic workload generator (Faban) to generate load against a realistic Web 2.0 application (Olio) [49]. The stack is deployed on Amazon EC2 instances. Faban is an open-source, Markov-chain, session-based synthetic workload generator, while Olio 2.0 is a two-tier social networking benchmark, with a web frontend and a database backend. The application metric is the number of active users of the social networking application, which drives the throughput or the number of operations per second [40].

2.6 Summary

While auto-scaling is a vital feature of any cloud environment, its necessity and nature are differ widely from IaaS to PaaS. Based on our analysis, we propose that proactive decision making and efficient, workload-aware resource allocation are two of the key features that can be expected from a good PaaS auto-scaler.

In our research on existing auto-scaling solutions, we have observed that most of the existing proposals and implementations focus on IaaS rather than PaaS. Recent developments like AppScale and IBM Bluemix make notable attempts to bridge this gap. While the Apache Stratos auto-scaler also shows a good potential, it has much room for improvement, particularly in the aspects of proactive decision-making and efficient resource management.

We have also explored existing approaches for workload prediction based on time series analysis and machine learning, and present some of the existing proposals for resource allocation from the perspectives of quality of service and cost. In addition, we have identified multiple workload experimentation platforms, workload generators, real and synthetic workloads, and application benchmarks, which would be useful for performance evaluations of an auto-scaling implementation.

Chapter 3

Evaluation of Apache Stratos Auto-Scaler

Apache Stratos is an open-source PaaS framework originally developed by WSO2 and currently being revived by the Apache Software Foundation. Stratos can be used to configure and deploy a PaaS on existing IaaS resources, which allows a customer to utilize his/her existing IaaS resources by encapsulating them to the level of reduced granularity and complexity of a PaaS platform. It also supports multiple IaaS providers, including AWS, OpenStack, GCE (Google Compute Engine) and Microsoft Azure (Microsoft Azure support is currently under development).

Current Stratos auto-scaler is based on policy-driven decisions, which performs workload prediction for small time windows (usually one minute) but does not utilize resource optimization approaches explicitly (all available resources are considered equal, and scale up/down operations arbitrarily pick resources to be allocated/deallocated). In this chapter we will be evaluating the performance of Apache Stratos against various workloads and analyze the correctness and efficiency of the current Stratos auto-scaler implementation.

3.1 Stratos Implementation

Current implementation of the auto-scaling process in Apache Stratos can be broken into two steps as:

1. Workload Prediction
2. Resource Allocation Process

3.1.1 Workload Prediction

An important capability of Apache Stratos is predicting immediate future load based on current health statistics. All cartridges, load balancers, and health monitors publish health status into CEP (complex event processor) via a real-time event bus. The CEP will aggregate all of these events and calculate the first derivative and second derivative for a given timeframe and pass the resulting streams to the auto-scaler via the reliable message bus. The auto-scaler uses motion equation to get the immediate (next minute) future load.

$$S = u + \frac{1}{2}at^2 \tag{3.1}$$

- s = predicted load

- u = first derivative (e.g., first derivative of current load average)
- t = time interval (e.g., one minute)
- a = second derivative (e.g., second derivative of current load average)

The formula is used to predict future workloads using the following metrics of the system deployed using Stratos:

1. CPU load average
2. memory usage
3. request in flight (number of pending requests)

3.1.2 Resource Allocation Process

Apache Stratos has several policies that govern the auto-scaling process which can be identified as:

1. Deployment Policy
2. Auto-Scaling Policy

In each of these policies, a set of parameters are provided by the user to calibrate the elasticity of the system. The auto-scaling policy and the deployment policy will specify the parameters that will be used for auto-scaling. The system will auto-scale according to the threshold values specified in the auto-scaling policy related parameters, and will also be aligned to the minimum and maximum partition size controls that are set through the deployment policy parameters.

Deployment Policy

Two parameters related to auto-scaling are defined in the deployment policy file as follows:

partitionMin It is possible to set a lower limit to the number of service instances that are maintained in the system at any given time by setting the partitionMin parameter for any service cluster. The auto-scaler will ensure that the system will not scale down below this value, even though there is no considerable service requests in-flight.

partitionMax The user can set the partitionMax parameter for any service cluster to control the maximum number of instances that the auto-scaler can start. The auto-scaler ensures that the system will not scale up above this limit, even when there is a high load of requests in-flight. As a result, when you have to pay for the instances that you start up, it is very useful to set the partitionMax parameter.

Auto-Scaling Policy

Three parameters related to auto-scaling are defined in the auto-scale policy configuration, as follows:

requestsInFlight The requestsInFlight parameter defines the number of requests the load balancer received, but are yet to be processed.

memoryConsumption The memoryConsumption parameter defines the memory usage of a cartridge process, which is considered as a percentage of the total available RAM. This is a percentage and cannot go beyond 100 since a process cannot take more than total physical memory.

loadAverage The loadAverage parameter defines the average system load over a period of time. (System load is a measure of the amount of computational work that a computer system performs.) This a percentage and can go beyond 100%.

Once the CEP computes the gradients and second derivatives for these parameters, the auto-scaler component will predict the workload of the system for the next minute, evaluate the predicted workload against the set of defined auto-scaling policies, and calculate the number of instances required to handle the workload ahead. As mentioned above, the Stratos auto-scaling process is implemented based on three metrics, hence the auto-scaling policy defines a threshold for each of these metric to calculate the optimal number of VMs required for each predicted metric, and the maximum predicted number of VMs among the predictions for all three metrics will be spawned if the number is greater than the currently active VM count, and vice versa.

3.2 Evaluation

In order to quantify the accuracy and effectiveness of this approach, we performed an evaluation for each of the two components of the auto-scaling workflow, using generated synthetic workloads and empirical workload obtained from logged traces. The prediction model was tested by simulating their mathematical model using drastically different workloads and the scaling process was tested by deploying Apache Stratos on AWS EC2.

3.2.1 Workload Generation

Synthetic workload is generated using a workload generation toolkit *Rain* developed by University of California at Berkeley [42] which is supplemented by an auction site prototype model *RUBiS*[46]. Empirical workload sets were obtained by several sources, including an open source human resource management system hosted on Rackspace cloud, and the university sports promoting site *Moraspirit* [18].

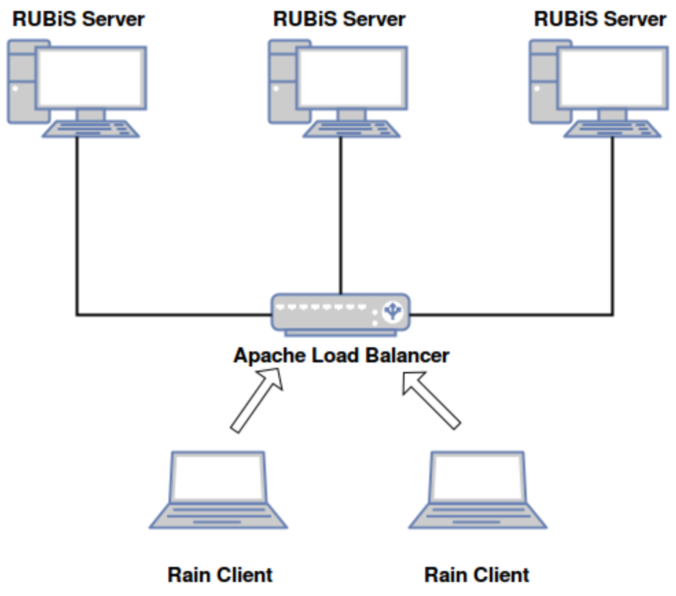


Figure 3.1: Simulation Setup

Synthetic Workload

To generate synthetic workloads we used the Rice University Bidding System (RUBiS) which is an auction site prototype modeled after the internet auction website eBay. To study the effect the different workload attributes on the response times, we performed a series of experiments by varying the workload attributes such as think time, inter-session interval, session length and number of users. We used the Rain workload generation toolkit to overcome the lack of flexibility and scalability in RUBiS client simulation. Using Rain we generated workloads by varying different attributes, targeted on the RUBiS system hosted at a different node. The experiments were performed on two machines with similar specifications: Intel Core i3 1.6 GHz processor, 4 GB memory and 500 GB hard disk capacity. We used a PHP incarnation of the RUBiS system for all our experiments. RUBiS was hosted on top of Apache 2.4.7 web server, complemented by MySQL 5.5.44 database server. Rain workload generator was run on top of Java 1.7 OpenJDK runtime environment. We used Sysstat 11.1.5 utility for measuring the system metrics.

Empirical Workload

In order to get the complete analysis we also used several real world workload traces from different sources. We generated workload data for requests in flight using log traces gathered from Moraspirit site and emulated the trace using httperf [39] on our simulation platform as shown in figure 3.1 which mirrored the Moraspirit site. We also collected data for memory consumption from an open source enterprise application hosted on a dedicated standalone linux server at Rackspace with Quad-Core AMD Opteron(tm) Processor 2374 HE and 16 GB memory capacity.

3.2.2 Evaluation of the Prediction

Accuracy of the prediction model used for auto-scaling is critical, since the scaling process is highly dependent on the predicted workload. Current implementation of Stratos uses a prediction model based on a mathematical model similar to the motion equation (3.1). We tested the accuracy of the model using various types of workloads using our simulation setup described above and chose a few workloads with significant characteristics that demonstrate the deficiency of the prediction model.

Case I - Constant Workload

We generated a high constant workload of 2000 users using Rain toolkit for the RUBiS benchmark and played it against our RUBiS PHP incarnation deployed on our simulation setup as described above.

Case II - Varying Workload

To emulate a real auction site we produced an exponentially growing workload starting from 20 users and reaching up to around 1200 users with rapid bidding and browsing actions.

Case III - Moraspirit Trace

We emulated the Moraspirit website behavior for 2 hours using 1 month duration of log traces by speeding up the dispatching of requests, and ran it on a mirrored copy of the Moraspirit application hosted on our simulation setup.

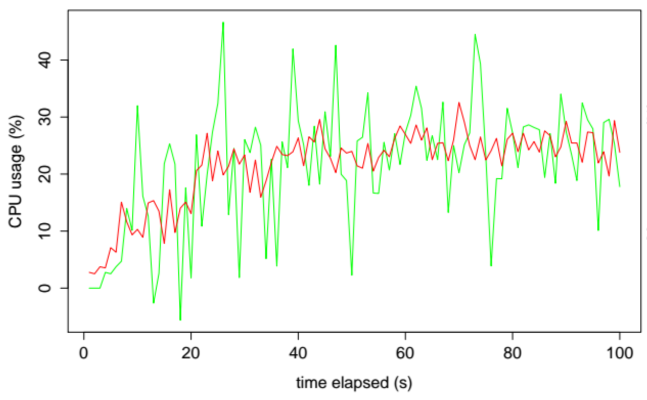


Figure 3.2: Load Average for Constant Workload

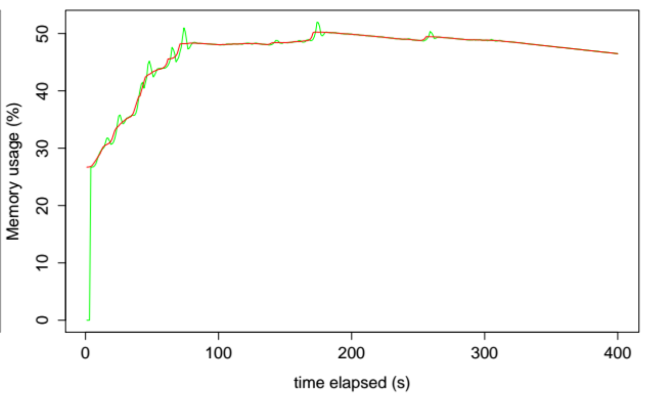


Figure 3.3: Memory Consumption for Constant Workload

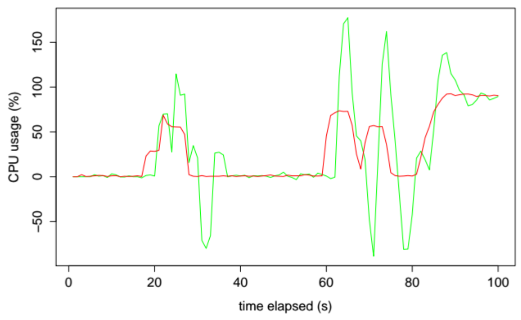


Figure 3.4: Load Average for Varying Workload

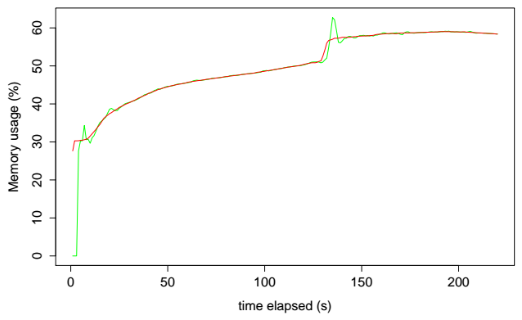


Figure 3.5: Memory Consumption for Varying Workload

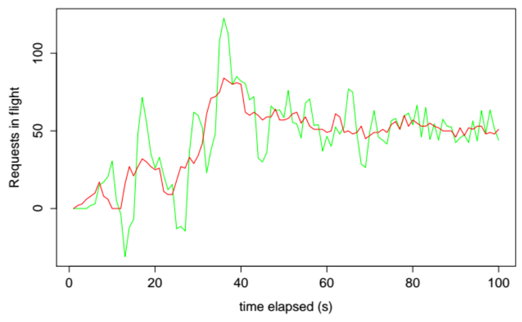


Figure 3.6: Request in Flight of Moraspirit Trace

Case IV - Enterprise Application Trace

We collected data for a period of 3 days at a frequency of 1 minute during the working hours where users would be accessing the system and operating on their usual day to day business activities.

As seen in the results above it is clear that the prediction model used in Apache Stratos cannot capture real-time workload variation patterns accurately and there is a significant lag in capturing such patterns. From the mathematical model itself, one can conclude that since its a second order polynomial it cannot really predict workload patterns of any higher orders.

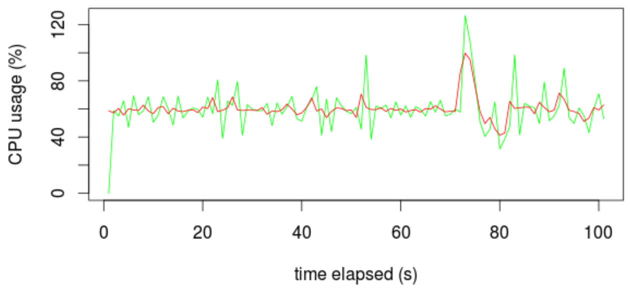


Figure 3.7: Load Average of Enterprise App

Though it is quite accurate in predicting the direction of the graph using the first and second derivatives, in real scenarios it almost always performs overpredictions with a significantly large margin, and hence a significant degree of unnecessary resource allocation.

3.2.3 Apache Stratos on AWS

In order to evaluate the scaling process of Apache Stratos we cannot simply use either a simulation or the built-in mock IaaS feature, since they do not capture every aspect of a real system such as fault detection and handling, spawn time delay and in-flight requests count. Hence we deployed Apache Stratos 4.1.4 on AWS EC2 in a distributed setup to run a few workloads using the RUBiS benchmark and allow the system to scale using default policies. Configuration details of the setup are given in table 3.1.

Table 3.1: Apache Stratos setup configuration

Stratos Component	Implementation	EC2 Instance Type	Details
Stratos	Apache Stratos 4.1.4	t2.medium	This includes Stratos Manager and the Auto-Scaler
Message Broker	Apache ActiveMQ 1.8	t2.medium	Used to communicate between each component and Cartridge Agent on each node
Load Balancer	HAProxy 1.6	t2.medium	One static load balancer to handle application requests
Complex Event Processor	WSO2 CEP 3.1	r3.large	Aggregates all health stats gathered from each node real-time and produces average, gradient, derivative values
Database	MySQL	r3.large	Central database for all nodes to read/write
Load Generator	Rain Tool Kit	r3.large	Creates multiple connections with the Load Balancer to emulate the stipulated users and their actions

At the time of evaluation, the latest stable version of Apache Stratos was 4.1.4. Apache ActiveMQ was chosen as it is the message broker recommended by Apache Stratos, although other brokers like RabbitMQ are also known to be supported. We tried almost all the options for the load balancer that Apache Stratos supported (according to their documentation), but we could only manage to set up HAProxy as the load balancer with a few modifications on our own. WSO2 CEP was the only choice for the event processor as the implementation is heavily coupled with the architecture of WSO2 CEP. We used a central database and scaled only the PHP instances, avoiding the trouble of syncing databases across multiple nodes which would have been the case if a MySQL was also provisioned on a cluster. For the central database we had two setups, one using AWS RDS and the other with our own dedicated server. Since AWS RDS had certain limitations such as the maximum number of concurrent connections, we set up our own node by tuning its performance to handle up to 5000 concurrent connections. Two extra nodes were set up on the same network to enumerate the workload for the RUBiS app deployed on top of Apache Stratos. A high-level diagram of the deployment is shown in figure 3.8.

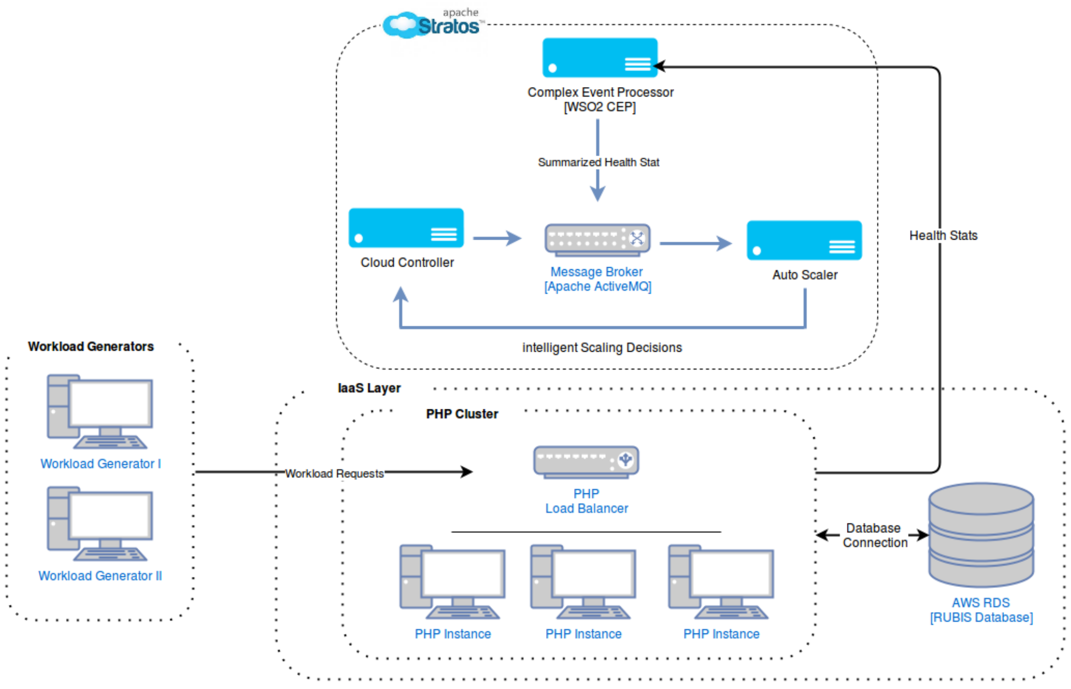


Figure 3.8: Deployment of Stratos on AWS EC2

3.2.4 Evaluation of the Scaling Process

Resource allocation and cost optimization are two important factors when it comes to auto-scaling at the PaaS level. In order to evaluate the effectiveness of the scaling process used in Apache Stratos, we deployed the RUBiS benchmark application on top of the Stratos AWS setup described above. We ran a single workload with varying resource demands for different thresholds to analyze the correctness of the scaling process. The workload is a browse-only type workload, defined for a period of for 15 minutes and cycled for a duration of 60 minutes with 30 seconds for rampup and rampdown as described below.

Table 3.2: Workload for Performance Analysis

Segment	1	2	3	4	5	6
Duration (seconds)	240	240	240	240	240	240
Users	400	800	1600	3200	800	400
Transition (seconds)	30	30	30	30	30	30

Case I - Low Thresholds

Using the workload described in table 3.2 for a low threshold values described in table 3.3

Table 3.3: Auto Scaling policy with low thresholds

load average	memory consumption	request in flight
70	70	100

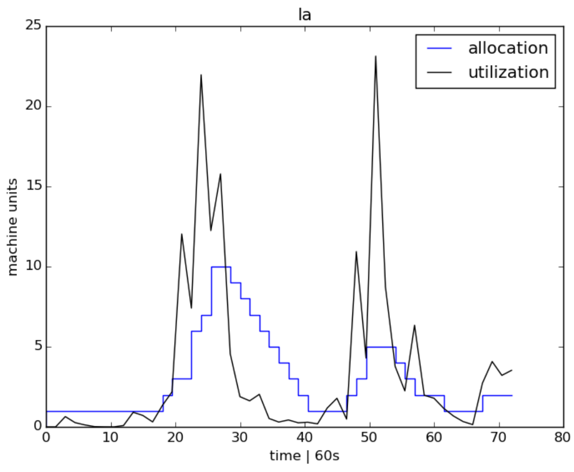


Figure 3.9: Load Average for Low Threshold

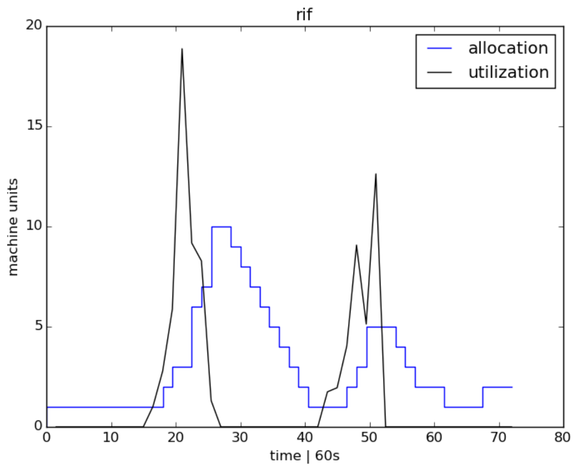


Figure 3.10: Request in Flight for Low Threshold

Case II - High Thresholds

Using the workload described in table 3.2 for a high threshold values described in table 3.4

Table 3.4: Auto-Scaling policy with high thresholds

load average	memory consumption	request in flight
95	95	150

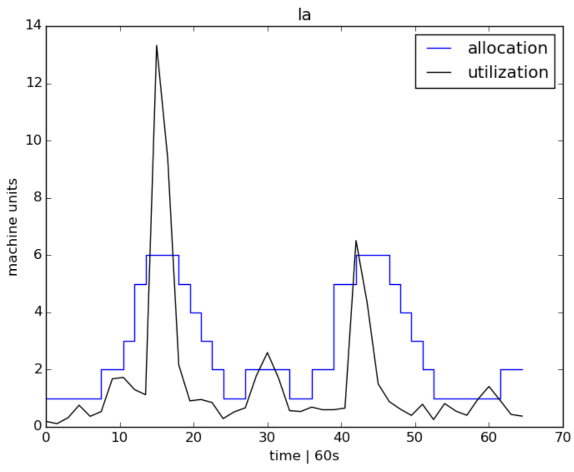


Figure 3.11: Load Average for High Threshold

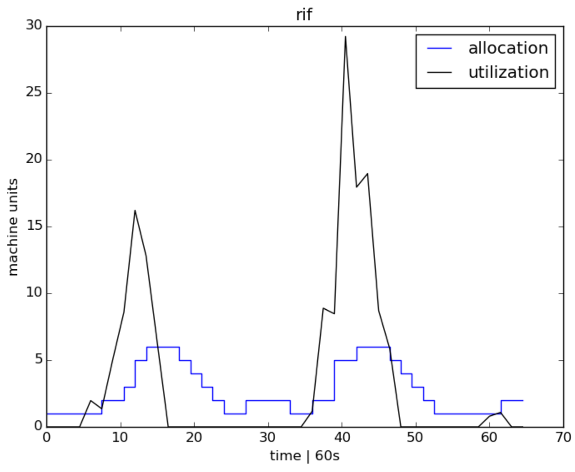


Figure 3.12: Request in Flight for High Threshold

The step curve depicts the variation of virtual machine count in the system over time (and hence the allocation of resources), while the jagged curve represents the variation of actual utilization of currently allocated resources. The latter is the product of machine count and the average cluster level utilization of each resource, resulting in a measure of the total fractional utilization of the resource in the cluster in terms of *VM units*. We use this unit to denote the fractional amount of machines required to be running at full capacity to fulfill a given load requirement.

The first test case indicates that, although violations of resource requirements are rare and compensated within relatively short periods of time, the actual load average of the system stays below the allocated amount over a major fraction of the time. This means that a significant portion of the allocated resources is in idle state.

The second test case obtained under higher thresholds indicates better levels of resource utilization (the utilization curve being closer to the allocation curve), meaning that resources are being allocated more efficiently. However, there are several cases of severe violations of resource requirements over quite noticeable periods of time. This is because the system is allowed to proceed without scaling up under high fractional loads, meaning that it would not have sufficient time to cater for an increasing workload before reaching a state of over-utilization.

Table 3.5: QoS summary for evaluation

test case	average response time	requests initiated	requests completed	time out errors
low threshold	6.3839 s	254535	203067	2511
high threshold	4.4954 s	280477	191891	11863

As shown in table 3.5 low threshold configuration has a better QoS delivery compared to the high threshold configuration. Low threshold configuration has a 80% successful request completion ratio where as high threshold configuration only has about 68%. Even though the average response time for low threshold is comparatively low, it is because the number of drop offs in high threshold is significantly higher. Connection timeout errors is also significantly larger in high threshold configuration where 4.2% account for the number of such errors whereas it is only 0.9% in low threshold configuration.

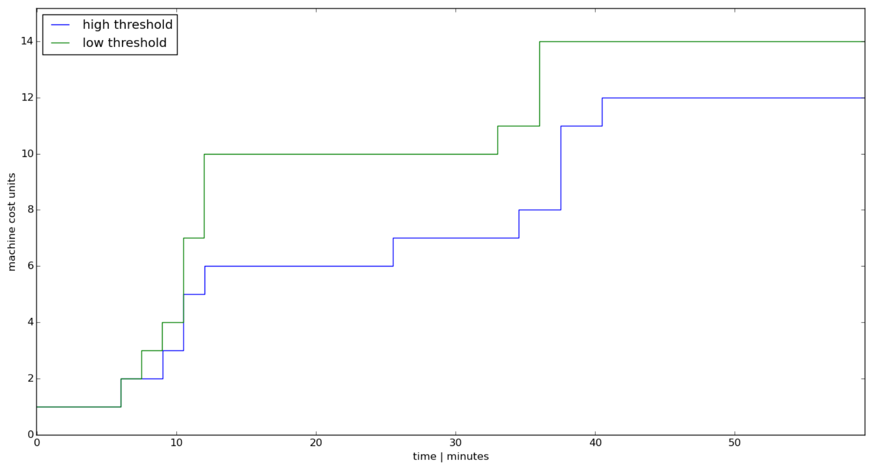


Figure 3.13: Cost comparison for high threshold configuration and low threshold configuration

Even though low threshold is comparatively a better configuration in terms of QoS, it also has a significant increase in cost compared to high threshold configuration. As shown in figure 3.13 low threshold configuration has a higher cost since it always spawn instances in a quick succession to meet the workload demand. The workload considered is only for a duration of one hour and comparatively high threshold configuration saves the cost for two machine units for degraded QoS. Hence the threshold value that needs to be chosen for the auto-scaling policy is tightly coupled with both QoS and operational cost and its not a trivial task for an average PaaS user to define such thresholds without prior domain knowledge.

3.2.5 Summary

At the platform-as-a-service (PaaS) level, auto-scaling mainly deals with infrastructure resource management and cost saving for the customer. Although meeting High Availability (HA) and other QoS parameters is important, the user also desires to consume just the right amount of resources necessary to satisfy the performance requirement of the application. Currently in Apache Stratos, scaling decisions are based on a rule engine which is inadequate for predicting workload or learning application characteristics, and also inefficient in resource allocation. The scaling process neither considers the cost involved in the scaling decision nor takes SLA factors into consideration. Following are the key defects of the current implementation of Apache Stratos:

1. Inaccurate prediction of workload
2. Non-triviality of setting up threshold values for applications with varying characteristics
3. Auto-scaling process not taking the cost factor into consideration
4. Blind killing resulting in inefficient resource allocation

Chapter 4

Workload Prediction

We have identified following challenges specific to workload prediction for auto-scaling:

- A PaaS cloud system may be used to build different applications with vastly different workload patterns. Hence, the workload prediction model should not tend to get overfitted to a specific workload pattern.
- As the workload dataset grows with time, the predictive model should evolve and continuously learn the latest workload characteristics.
- The workload predictor should be able to produce results within a bounded time.
- Given that the time horizon for the prediction should be chosen based on the physical constraints like uptime and graceful shutdown time of Virtual Machines (VMs), the predictor should be able to produce sufficiently accurate results over a sufficiently large time horizon.

The objective of this section is to come up with a prediction method that can be trained in real-time to capture the latest trends and provide sufficiently accurate results for drastically different workload patterns. First, we evaluate the ability of existing models to produce accurate real-time predictions against evolving datasets. In this evaluation we tested time series forecasting methods, including statistical methods like ARIMA and exponential model, machine learning models like neural networks, and a prediction method used in Apache Stratos, an open source PaaS framework. Through this, we demonstrate the limitations in existing solutions in accurately predicting real-world workload traces, and highlight the need for more accurate predictions. Next, we propose an ensemble technique which combines results from a neural network, ARIMA and exponential models. Based on simulations with three publicly available workload traces (two real-world cloud workload datasets and the Google cluster dataset), we demonstrate that the proposed ensemble model outperforms each of the tested individual models.

4.1 Evaluation of Existing Models

In this analysis, we quantitatively and qualitatively analyze the ability of several prediction techniques to learn from the current workload history, and then predict future workload while the workload itself is evolving, similar to what happens in case of an auto-scaler in operation. We simulated an online training scenario by streaming data points of the workload time series one at a time to each prediction technique. We then plotted the one-step look-ahead predictions from the resulting prediction technique at each stage, against the real data points. For evaluation purpose, we used the forecast package in R [50] which contains time series data-sets in different domains with typical patterns like trends, cycles and seasonality factors.

Section 3.1 presents the workloads used for the evaluation while Section 3.2 describes the time series prediction techniques selected for the evaluation. Simulation setup is presented in Section 3.3. Evaluation of selected techniques under selected workloads is presented in Section 3.4.

4.1.1 Datasets

As the objective of this experiment is to identify how well an individual prediction method can predict future values by only considering the past history, we used several publicly available datasets with predictable patterns from the R forecast package, some of which are not directly related to cloud workloads. Selection of multiple datasets enables us to evaluate the time series prediction techniques under several probable workload patterns. Neverthless, the experimental results in Section 5, where we compare the performance of the proposed ensemble technique vs. existing methods, are based only on two real-world cloud workloads and the Google cluster dataset, in addition to those found in the forecast package.

Following datasets from the R forecast package is selected for the evaluation:

- euretail – Quarterly retail trade index in the Euro area (17 countries) from 1996 - 2011, covering wholesale and retail trade, and repair of motor vehicles and motorcycles (Index: 2005 = 100).
- sunspotarea – Annual averages of the daily sunspot areas (in units of millionths of a hemisphere) for the full sun.
- oil – Annual oil production (millions of tonnes) from Saudi Arabia between 1965 - 2010.

In addition, as a representative of real world workloads, we have used a CPU usage trace obtained from a dedicated standalone server from a private cloud. The server has a Linux-based operating system and is equipped with four Quad-Core AMD Opteron CPUs and 16 GB of RAM. We have also obtained a trace of queued HTTP requests over time by replaying an access log of a content-based website against a hosted copy of itself.

4.1.2 Time Series Forecasting Techniques

We chose three widely used prediction methods from literature, namely ARIMA, neural network, and exponential models, as well as the existing workload prediction technique in Apache Stratos.

ARIMA

Autoregressive Integrated Moving Average (ARIMA) model combines two models called autoregression (of order p) and moving average (of order q).

Autoregression Model AR(p) – In an autoregression model, the variable of interest is forecast using a linear combination of past values of the variable. The term autoregression indicates that it is a regression of the variable against itself. Thus, an autoregressive model of order p can be written as:

$$x_t = c + \phi_1 x_{t-1} + \phi_2 x_{t-2} + ... + \phi_p x_{t-p} + e_t \tag{4.1}$$

where c is a constant and e_t is white noise (list of Symbols are given in Table 4.1). Autoregressive models are typically restricted to stationary data, and some constraints are applied on the values of the parameters [17].

Moving Average Model MA(q) – This model uses past forecast errors in a regression-like model.

$$x_t = c + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_q e_{t-q} \tag{4.2}$$

where e_t is white noise. x_t can be thought of as a weighted moving average of the last few forecast errors. By combining differencing with autoregression and a moving average model, non-seasonal ARIMA model can be obtained. The full model can be written as:

$$\begin{aligned} x'_t = c + \phi_1 x'_{t-1} + \phi_2 x'_{t-2} + ... + \phi_p x'_{t-p} \\ + e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_q e_{t-q} \end{aligned} \tag{4.3}$$

The *predictors* on the right hand side include both the lagged values of X_t and lagged errors. This is called an ARIMA(p, d, q) model, where p and q are the orders of the autoregressive and moving average parts respectively, and d is the degree of first differencing involved.

Neural Networks

Artificial Neural Networks (ANNs) are a class of nonlinear, and data-driven models which are inspired by the working mechanism of human brain. In the domain of time series forecasting, neural network is an excellent alternative for existing statistical models which require some assumptions to be satisfied in the time series. Ability of modeling nonlinear relationships is the major advantage in neural networks. Depending on the topology of the network there are various types of neural networks which are suitable for various machine learning tasks.
Auto regressive feed forward neural networks [17] are the specialized neural networks in the time series forecasting domain. They consist of a feed forward structure of three types of layers, an input layer, one or more hidden layer, and an output layer. Each layer consists with nodes, which connected to those in the immediate next layer by acyclic links which have associated weights. In auto regressive neural networks, lagged values of the time series (input window) are injected as the inputs and train the weights of the model to forecast for a given time horizon by using the past history of the dataset.

Exponential Model

Several selected versions of exponential models are presented next.

Exponential Weighted Moving Average (EWMA) – In EWMA methods, forecasts are calculated using weighted averages where the weights decrease exponentially as the observations come from further in the past. The smallest weights are associated with the oldest observations [17], [51], [52]. Exponential smoothing can be indicated by the following equation:

$$\hat{x}_{t+1|t} = \alpha x_t + \alpha(1-\alpha)\hat{x}_{t|t-1} \tag{4.4}$$

$$\hat{x}_{t+1|t} = \alpha x_t + \alpha(1-\alpha)x_{t-1} + \alpha(1-\alpha)^2 x_{t-2} + ... \tag{4.5}$$

where $0 \leq \alpha \leq 1$ is the smoothing parameter. $\hat{x}_{t+1|t}$, the one-step-ahead forecast for time $(t+1)$, is a weighted average of all the observations in the series x_1, ..., x_t. The rate at which the weights decrease is controlled by the parameter α. If α is small (i.e., close to 0), more weight is given to observations from the distant past. If α is large (i.e., close to 1), more weight is given to the more recent observations. This method is suitable for forecasting data with no trend or seasonal patterns.

Following variations of basic exponential smoothing are suitable for forecasting time series with various characteristics [17]:

- *Double Exponential Smoothing* – Applicable to a time-series with a linear trend.

Table 4.1: List of Symbols in Stratos Prediction Model

Parameter	Description
$S_{t+h\|t}$ x	Predicted change of workload for time $(t+h)$ based on interval t Workload metric(s)
$\hat{x}_{t+h}$	Predicted workload metrics for time $(t+h)$
u_t	First derivative of the workload metrics within the t-th interval
a_t	Second derivative of the workload metrics within the t-th interval
h	Time horizon

- *Triple exponential smoothing* – Applicable to a time-series with a trend and seasonality.
- *Holt's Linear Trend Model* – Extended simple exponential smoothing to allow forecasting of data with a trend.
- *Exponential Trend Model* – Applicable when the trend in the forecast function is can be exponential rather than linear.
- *Damped Trend method* – Dampens the trend to a flat line some time in the future to model the general nature of practical time series.
- *Holt-Winters Seasonal method* – Applicable to a time series with adaptive or multiplicative trends and seasonality.

Prediction method in Apache Stratos

Apache Stratos is an open source PaaS framework where a user can build a PaaS system on top of an existing IaaS cloud. The auto scaling solution in Stratos uses a workload demand prediction method which can be explained using the motion equation in physics. The method calculates one lookahead prediction using the workload demand in the last period as follows [2], using symbols in Table 1:

$$S_{t+h|t} = u_t h + \frac{1}{2} a_t h^2 \tag{4.6}$$

$$\hat{x}_{t+h} = avg(x)_t + S_{t+h|t} \tag{4.7}$$

4.1.3 Time Series Forecasting Library

We used the following time series forecasting models in R forecast package [50] to implement each of the selected forecasting solution:

- *auto.arima*() – Finds out the best fitted ARIMA model and approximates the model coefficients by minimizing the past forecast error [17].
- *ets*() – Finds out the best fitted exponential model among the family of exponential models (simple exponential smoothing, Holt's linear trend, exponential trend, and Holt-Winters seasonal method) automatically and calculates the model coefficients by reducing the past forecast error [17].
- *nnetar*() – Feed-forward neural networks with a single hidden layer and lagged inputs for forecasting uni-variate time series.

4.1.4 Error Measures

For quantitative analysis, we used the following two error measures:

- *Root Mean Squared Errors (RMSEs)* are of the same scale as the data. As it is scale dependent, it cannot be used to make comparisons between series that are of different scales [17]. RMSE can be represented as follows:

$$RMSE = \sqrt{\frac{\sum_{t=1}^{n}(\hat{x}_t - x_t)^2}{n}}$$

  where x_t is the real value at time t, $\hat{x}_t$ is the forecast value at time t and n is the number of data points in dataset.

- *Percentage errors* have the advantage of being scale-independent, but they overemphasize the errors for values that are actually small. Percentage errors can be represented as follows:

$$MAPE = \frac{100}{n}\sum_{t=1}^{n}\left|\frac{\hat{x}_t - x_t}{x_t}\right|$$

  where x_t is the real value at time t, $\hat{x}_t$ is the forecast value at time t and n is the number of data points in dataset.

4.1.5 Observations

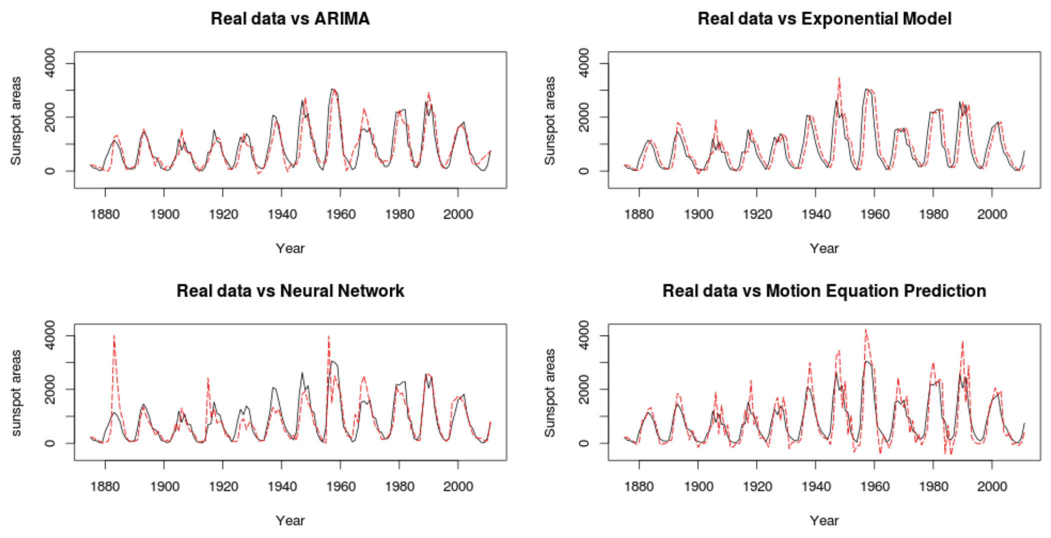


Figure 4.1: Comparison of predictions for sunspotarea dataset.

Figures 1 to 4 show the comparison of each of the time series prediction models under four datasets. Solid lines represent the actual time series whereas dashed lines represent the predicted values from each method. The respective error measures are presented in Tables 2 and 3. The highlighted cells represent the minimum RMSE and MAPE values.

According to the results obtained, there is no single model which performs well in all online training scenarios. Each model will fit the datasets which satisfy its assumptions. Where they are not satisfied, performance of some models would be below the average. For example, while neural networks perform well on the *oil* dataset, they perform poorly on *euretail* dataset. The current prediction method from Stratos has the highest mean errors almost all the cases considered.

The analysis primarily depicts that there is no single method which guarantees best performance in all cases. A method which performs well in some dataset might perform worse in another dataset. As an autonomous PaaS auto-scaler should be able to work with any type of

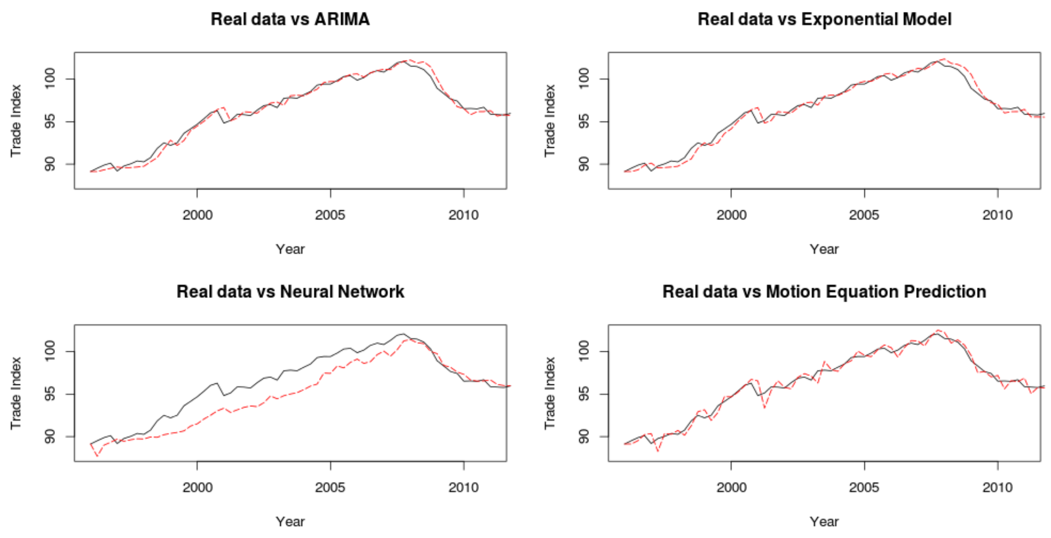


Figure 4.2: Comparison of predictions for euretail dataset.

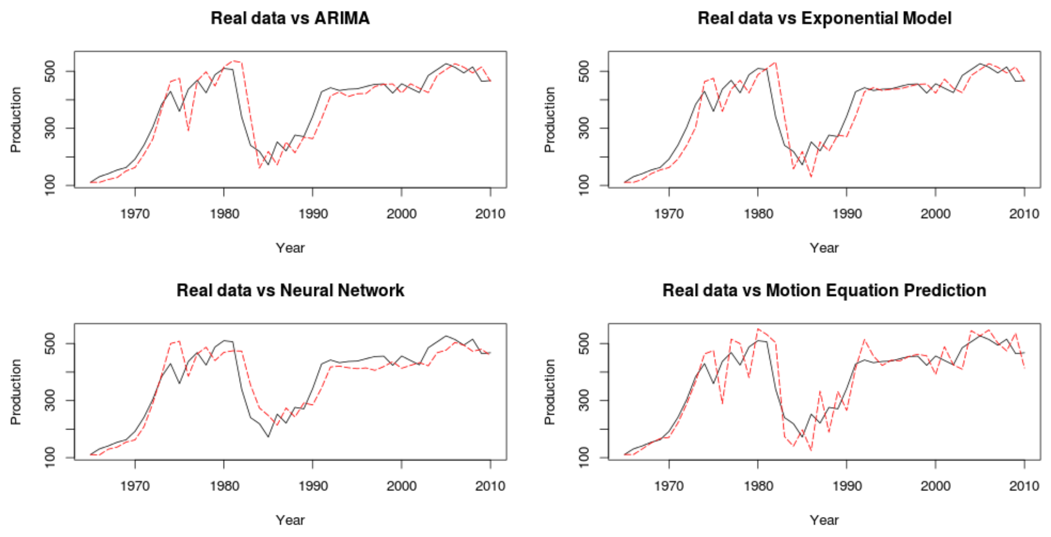


Figure 4.3: Comparison of predictions for oil dataset.

workload pattern, the prediction method should be able to give good enough estimates in an average case without producing large errors on any specific workload pattern.

The idea of ensemble learning is quite frequently used in situations where there is no dominant technique which can provide the best results, but it is possible to obtain good enough results by combining results from several weak learners. In the general forecasting domain we can see several models combining techniques proposed by several researchers [53], [54], [55].

There are several views regarding model selection and combining multiple models. While some researchers claim that combined methods improve accuracy, others claim that they suppress the effects of large errors from individual models rather than improving the overall accuracy. According to the literature, there are multiple ways of combining individual results [56]:

- Simple average – Obtaining the average of the forecast from each model as the forecast of ensemble.
- Median-based – Obtaining the median of the forecast from each model as the forecast of ensemble.

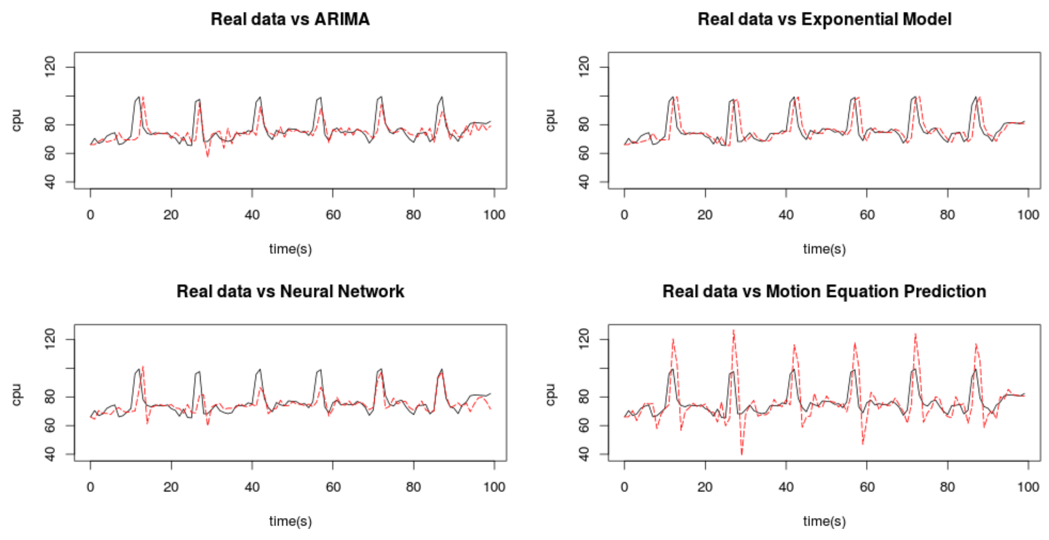


Figure 4.4: Comparison of predictions for CPU utilization of private cloud server.

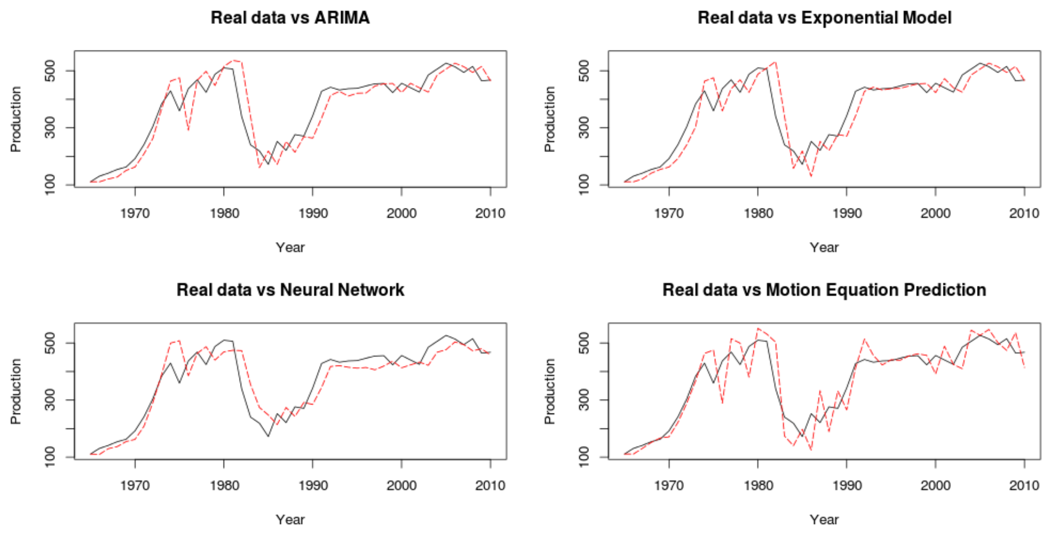


Figure 4.5: Comparison of predictions for queued HTTP request count.

- Trimmed average – Obtaining the average of forecasts while excluding the worst performing models.
- Error-based combining – Assigning the weight of each model to be the inverse of the past forecast error (e.g., MAE, MAPE, or RMSE) of each model.
- Variance-based method – Determining the optimal weights through minimization of the total sum of squared error.

4.2 Proposed Ensemble Prediction Method

We propose an error-based ensemble technique for workload prediction. Our approach tries to address situations where offline training is not possible due to workload history data not being available at the beginning of the prediction process. While the auto scaler is in operation, workload history gets accumulated based on the user's workload requirement (e.g., CPU, memory, and request count). However, the prediction method should still be able to predict the future time horizon based on the initially available dataset. After the initial predictions, actual data will become available for the next time periods, so that we can accumulate the latest actual

Table 4.2: Comparision of errors for standard datasets using RMSE and MAPE

Model	sunspotarea		Euretail		oil	
	RMSE	MAPE	RMSE	MAPE	RMSE	MAPE
ARIMA	**382.360**	$\underline{1.359}$	**0.524**	**0.004**	55.313	0.251
Exponential	505.750	1.161	0.576	$\underline{0.011}$	54.989	0.250
Neural net.	473.924	**0.465**	$\underline{1.882}$	0.006	**51.616**	**0.160**
Current	$\underline{546.938}$	0.965	0.650	**0.004**	$\underline{61.807}$	$\underline{0.585}$

Table 4.3: Comparision of errors for cloud datasets using RMSE and MAPE

Model	RIF		CPU	
	RMSE	MAPE	RMSE	MAPE
ARIMA	7.238	0.136	2.976	0.036
Exponential	**7.005**	0.160	3.150	$\underline{0.048}$
Neural net.	8.169	**0.135**	**2.792**	0.031
Stratos	$\underline{9.928}$	$\underline{0.172}$	$\underline{5.692}$	**0.024**

data into the workload history and use them for the next forecast horizon.

In existing error-based combining techniques, mean values of error metrics (e.g., absolute error, absolute percentage error, squared error, etc.) are taken into account while calculating the contributing factors for individual methods.

Considering the currently available dataset in the time series as $X = [x_1, x_2, ..x_t]$ we want to calculate predictions for x_{t+h}. Let the final predicted value be $\hat{x}_{t+h}$ and the predicted value from the i-th model for x_{t+h} be $\hat{x}^{(i)}_{t+h}$. We define $\hat{x}_{t+h}$ as a weighted sum of predictions from each model:

$$\hat{x}_{t+h} = \sum_{i=1}^{k} w_i \hat{x}^{(i)}_{t+h} \quad \forall k \in \{1, 2, 3, ..., n\} \tag{4.8}$$

where w_i is the weight assigned to the i-th forecasting method. To ensure unbiasedness, it is often assumed that the weights add up to unity [56]. Hence, we define the contribution from the i-th model to the final result as c_i, so that the same equation can be redefined as:

$$\hat{x}_{t+h} = \frac{\sum_{i=1}^{k} c_i \hat{x}^{(i)}_{t+h}}{\sum_{i=1}^{k} c_i} \tag{4.9}$$

where $w_i = \frac{c_i}{\sum_{j=1}^{k} c_j}$. As $\sum_{j=1}^{k} w_j = 1$, this weight assignment is unbiased and would result in the weighted average of the predictions.

4.2.1 Determination of Contribution Coefficients

To determine the optimal contribution coefficients in our proposed ensemble technique, we use past forecast errors for each model. In our prediction problem, accuracy of the prediction of the next time horizon is more significant. Hence, the contributions are calculated using inverses of the past forecasting error measures. There are various choices for error measurement, and next we discuss several selected measures.

Error of the Most Recent Prediction

This model only captures how accurately the last prediction has been calculated by the model. Hence, it does not capture the overall accuracy. If the model has repeatedly made large errors

in past predictions, except the most recent one, contribution coefficients are biased and may lead to error-prone decisions.
Absolute Error of the Last Prediction can be defined as follows:

$$AE_t = |\hat{x}_t - x_t|$$

Squared error of the Last Prediction, which penalizes errors more efficiently than the absolute error, can be defined as follows:

$$SE_t = (\hat{x}_t - x_t)^2$$

Whereas *Absolute Percentage/Relative Error* is sensitive to errors related to values of small magnitudes and is defined as:

$$APE_t = 100\left|\frac{\hat{x}_t - x_t}{x_t}\right|$$

Average Error Over Prediction History

This measure captures the overall error in past predictions, where all past errors have the same level of significance. If a method has produced larger errors in past predictions, the contribution may be reduced significantly even though it may be producing the best predictions for more recent values.
In this case *Mean Absolute Error/Root Mean Squared Error* can be defined as:

$$RMSE = \sqrt{\frac{\sum_{t=1}^{n}(\hat{x}_t - x_t)^2}{n}}$$

Whereas *Mean Absolute Percentage Error* is defined as:

$$MAPE = \frac{100}{n}\sum_{t=1}^{n}\left|\frac{\hat{x}_t - x_t}{x_t}\right|$$

While calculating the contributions based on errors, models whose errors are based on the last observation overlook overall accuracy and assign high importance to the last prediction error. On the other extreme, the average errors assume that all the last prediction errors have the same level of significance. However, what we really need is an error measure which has a larger contribution from the errors in more recent predictions and smaller contributions from early predictions. Exponential smoothing provides the best fit for our requirement, so that we can calculate the contribution as the errors are fitted under the exponential model. Here, the contribution coefficients $c_{i,t}$ are calculated from the inverses of the the fitted past forecast errors (e_t). If $b_{i,t}$ is the fitted value of the past forecasting error from the i-th model at the t-th time interval, $c_{i,t} = \frac{1}{b_{i,t}}$. e_t can be the absolute error, squared error, or absolute relative error at t-th prediction. $b_{i,t}$ can be defined as follows:

$$b_{i,t} = \alpha e_{(i,t)} + (1-\alpha)b_{(i,t-1)} \tag{4.10}$$

where $0 \leq \alpha \leq 1$

$$b_{i,t} = \alpha e_{(i,t)} + \alpha(1-\alpha)e_{(i,t-1)} + \alpha(1-\alpha)^2 e_{(i,t-2)} + ...$$

$$c_{i,t} = \frac{1}{b_{i,t}} \tag{4.11}$$

4.2.2 Selection of Models

The above mentioned error-based weighting mechanism can be used with different combinations of forecasting models. To preserve the ability to cope up with drastically different workload patterns, models used in ensemble techniques should conceptually be different and capture the different characteristics of datasets.

ARIMA assumes a linear form of the model, i.e., a linear correlation structure is assumed among the time series values. Therefore, it cannot capture non-linear patterns [54]. Alternatively, seasonal ARIMA models can fit the seasonal factors of the time series accurately.

A neural network can capture complex nonlinear relationships within a time series. It has a data driven approach which can extract patterns within a time series without predefined knowledge on the relationships inside data. But neural networks require sufficiently large amounts of data points to accurately identify patterns. Observations in Section 3 also show that neural networks make larger errors in initial stages due to lack of data, but perform well after they correctly identify the relationships. Time to train a neural networks is also significant factor. As we propose real-time training, the data point history should not be allowed to grow arbitrarily large, which would slow down the prediction process.

Contrary to the general belief that ARIMA is more general than exponential models, there are no equivalent ARIMA models for some of the nonlinear exponential models (e.g., the exponential trend model) [17]. Therefore, to preserve generality, we also choose the exponential model as part of our ensemble solution.

There is a possibility that the above models may produce out-of-range forecasts, especially in the case of early-stage neural networks. To compensate for this, we use a naive forecast model which forecasts the last known data point for the next interval as well.

4.2.3 Proposed Prediction Algorithm

Proposed workload prediction technique can be explained using the following algorithm:

1. Consider the time series history window at time t, $X = [x_1, x_2, ..x_t]$.
2. Calculate forecast value from the i-th time series forecasting method over the time horizon h, $\hat{x}^{(i)}_{t+h} \forall i \in \{1, 2, 3, ..., k\}$, where k is the number of forecasting methods used.
3. Fit a history window for the last t actual data points using the i-th method.
4. Use an exponential smoothing model to fit the errors resulting from Step 3, and use them to calculate the contribution factor for the i-th model at t, $c_{(i,t)}$.
5. Calculate the point forecast for time $(t + h)$ using $\hat{x}_{t+h} = \frac{\sum_{i=1}^{k} c_{(i,t)} \hat{x}^{(i)}_{t+h}}{\sum_{i=1}^{k} c_{(i,t)}}$.
6. At time $(t+1)$, actual value for time $(t+1)$ will be available. Add this value to the history window $X = X \cup \{x_{t+1}\}$ and repeat from Step 2.

4.3 Experimental Results

We implemented the proposed ensemble-based prediction algorithm in R using time series and machine learning model implementations in the forecast package. For our ensemble solution we selected ARIMA, Neural Network, Exponential Model and Naive Prediction as the base models. After emperical evaluation of various error measurements for contribution coefficient

Table 4.4: Comparison of performance on standard datasets.

The ARMA-based model has been taken from [4].

Model	sunspotarea		euretail		oil	
	RMSE	MAPE	RMSE	MAPE	RMSE	MAPE
ARIMA	382.360	1.359	**0.524**	0.004	55.313	0.251
Exponential	505.750	1.161	0.576	0.011	54.989	0.251
Neural net.	473.924	0.465	1.882	0.006	51.616	**0.160**
Stratos	546.938	0.965	0.650	0.004	61.807	0.585
ARMA-based model	530.160	1.181	0.600	0.004	51.986	0.250
Mean ensemble	369.525	0.519	0.697	**0.003**	**49.922**	0.227
Median ensemble	397.439	0.777	0.531	0.004	50.046	0.250
Proposed ensemble	**356.315**	**0.393**	0.537	**0.003**	50.147	0.256

Table 4.5: Comparison of performance on cloud datasets

Model	Google Cluster		RIF		CPU	
	RMSE	MAPE	RMSE	MAPE	RMSE	MAPE
ARIMA	12.963	0.051	7.238	0.136	2.976	0.036
Exponential	12.886	0.041	7.005	0.160	3.150	0.048
Neural net.	12.530	0.036	8.169	0.135	**2.792**	0.031
Stratos	19.757	0.116	9.928	0.172	5.692	**0.024**
ARMA-based model	12.549	0.069	7.185	0.180	3.477	0.023
Mean Ensemble	12.099	0.051	7.036	0.130	2.900	0.029
Median Ensemble	12.059	0.055	7.010	0.141	2.944	0.028
Proposed Ensemble	**11.934**	**0.027**	**6.972**	**0.129**	2.873	0.027

calculation, we used square root of the exponentially fitted squared forecasting error as the error measure for calculating the contribution coefficients from each model (where $e_{i,t}$ is defined as $e_{(i,t)} = (x_t - \hat{x}_t^{(i)})^2$ in Equation 10 and $c_{(i,t)} = \frac{1}{\sqrt{b_{(i,t)}}}$ in Equation 11).

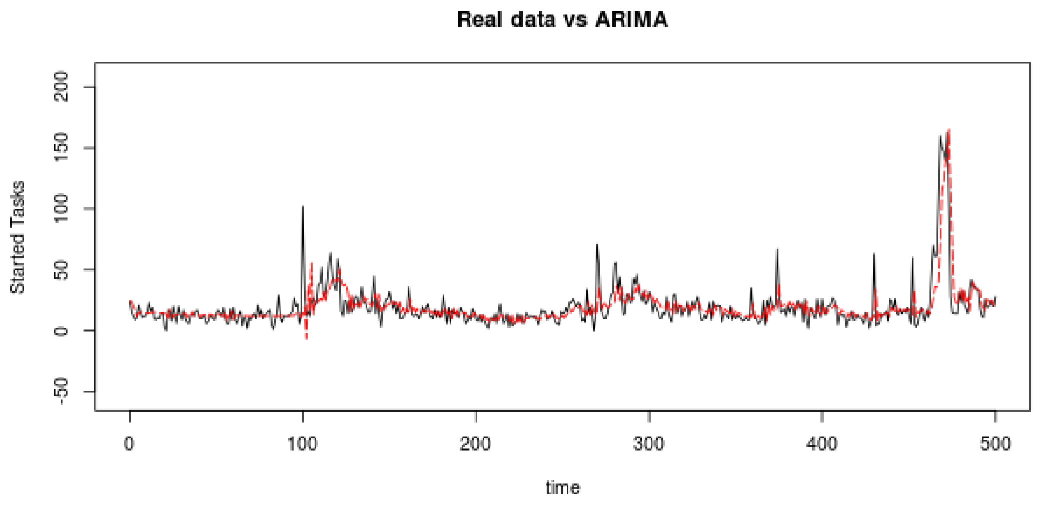
We tested proposed technique against the publicly available datasets mentioned in Section 4.2, as well as several real-world cloud workloads [18] collected from server applications. Our proposed solution is compared with each individual models in ensemble solution, two other popular ensemble techniques (mean ensemble, median ensemble), existing prediction technique in Apache Stratos and prediction model describes in [4].

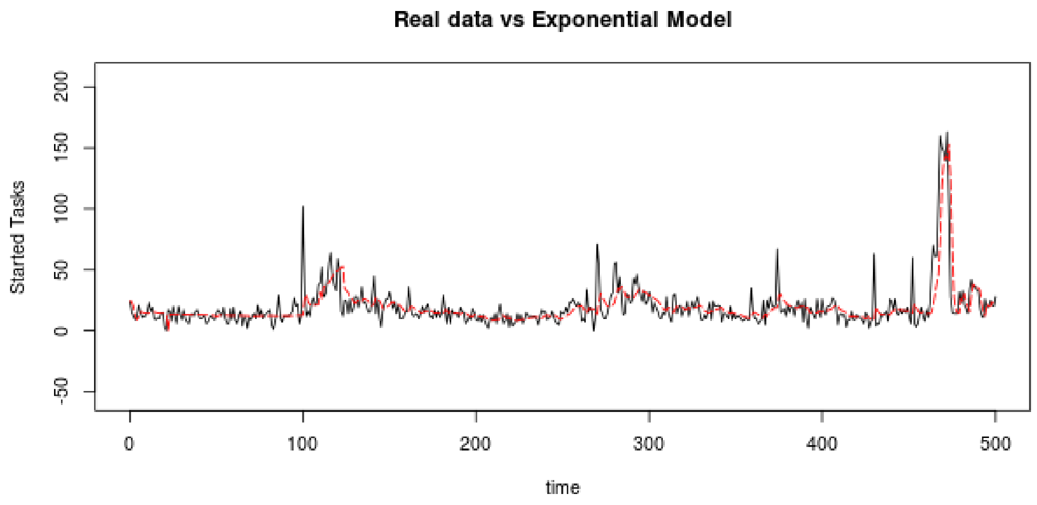
In addition to the cloud workload dataset used in Section 3, we also used a portion of the Google cluster dataset [45] in evaluating our ensemble model. The dataset includes a series of task execution details against their starting and ending times. Due to the large size of the dataset, we have summarized it to a suitably concise set of values by summing up numbers of started tasks in each time unit over the transformed time scale.

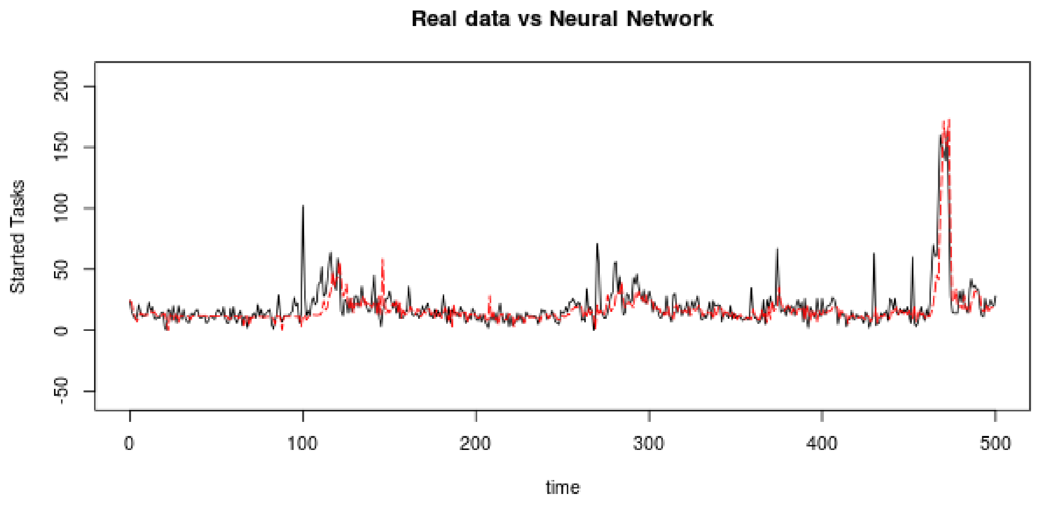
Boldfaced cells in Tables 4 and 5 contain the smallest error values for each dataset under evaluation, whereas underlined cells contain the worst performance observed for each dataset among all methods.

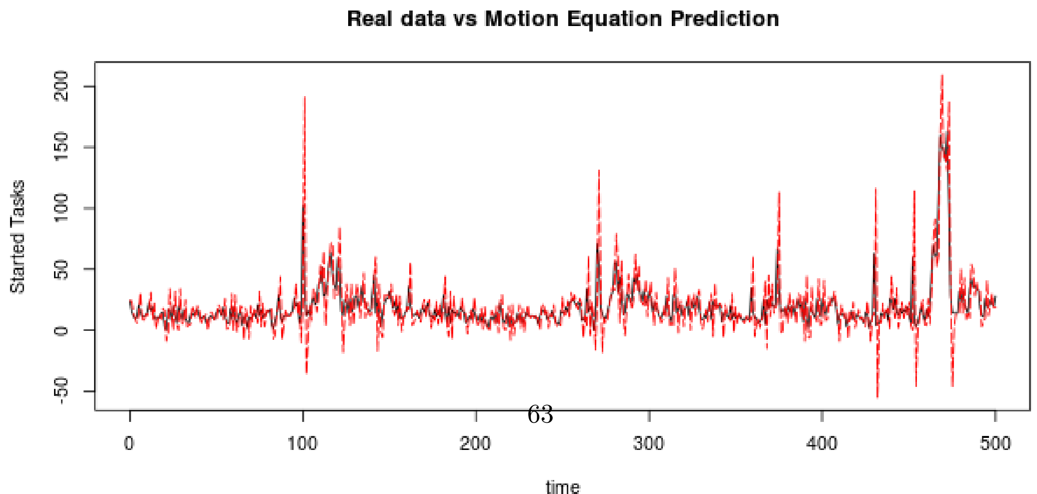
Figure 4.6 represents how each individual forecasting model fits the Google cluster dataset. It can be observed that the first three models show smoothened predictions, while the Stratos prediction model generates rapid fluctuations and values that are significantly off the range relative to actual values. As shown by Figure 4.7, although the ensemble model shows smoothened predictions at the start, it eventually manages to capture finer details of the series as the available dataset grows during real-time training.

According to the results obtained, the existing model from Stratos suffers from errors of the


Real data vs ARIMA
Started Tasks
time


Real data vs Exponential Model
Started Tasks
time


Real data vs Neural Network
Started Tasks
time


Real data vs Motion Equation Prediction
Started Tasks
time

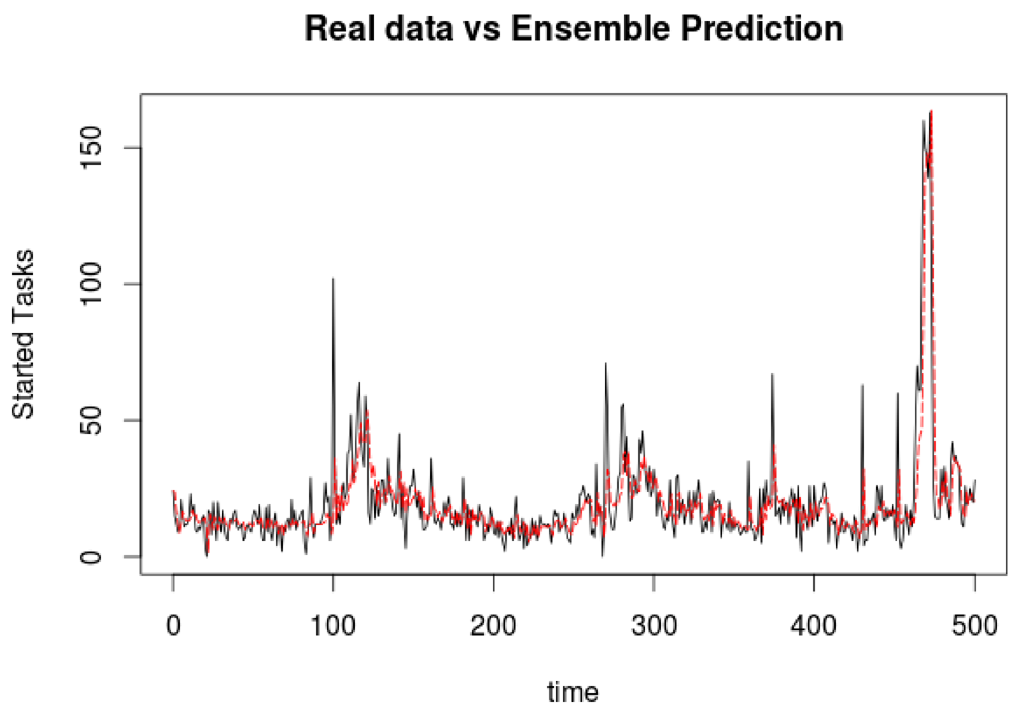


Figure 4.7: Ensemble prediction applied to Google cluster dataset.

largest magnitude in several datasets, while each individual model shows largest errors in at least one dataset.

While the mean and median ensemble methods generally perform better than individual models, they suffer when one or more of their base models perform significant errors, due to the lack of consideration of such forecasting errors resulting from individual models.

Our proposed prediction technique provides the best prediction results for several datasets while never performing worse than the individual prediction methods, because the contribution from each of its base models is explicitly governed by its accuracy.

Chapter 5

Resource Allocation

As identified in Chapter 3 auto scaling solutions has two major challenges. First, auto scaling mechanism should be able to predict the future workload. Chapter 4 described ensemble workload prediction mechanism based on machine learning and time series methods. Second challenge is to decide right number of resources based on the anticipated workload. In this chapter we discuss methods to allocate resources to anticipated workloads in a cost-effective manner while preserving QoS goals of the application.

5.1 Scaling Algorithm

Most of the public PaaS solutions like Google App Engine provide SLAs based on the QoS provided. According to these SLAs, a penalty will be charged from the provider to the PaaS user in a case where the provider was unable to provide the agreed level of service. In most cases this penalty is defined as a percentage of the monthly cost of users.

We propose a greedy heuristic scaling algorithm inspired by already existing PaaS SLAs. However, unlike most PaaS providers, we do not consider service uptime as the measure of QoS. Instead we consider any performance degradation with respect to the metrics considered (Memory Consumption, CPU utilization, Requests in Flight) as a violation. These different metrics affect the QoS of application in different ways.

1. Request In Flight (RIF)
   RIF is the amount of work that is pending to be delivered to a endpoint. Therefore it can be used to measure the expected future workload.It can also be thought of as a QoS measure as it is correlated to the latency of a a system. Higher the RIF value higher the number of requests queued to delivered. Therefore each request take higher time increasing the latency.In order to improve the latency factor system need to minimize the time number of requests in the queue. Only way to support this at platform level is to increase the number of endpoints that can serve requests.That is scaling up.

2. Memory Consumption
   Memory consumption shows how much memory application consume in currently. Therefore it can be considered as a measure of current workload. Failure to provide required amount of memory at platform would increase application latency as well as system throughput badly affecting the system QoS. Further it is possible that lack of memory causes system crashes affecting the service availability.

3. CPU utilization
   CPU utilization shows how busy the system processors. Therefore it can be considered as a measure of current workload. In QoS perspective CPU utilization can be interpreted

in the same way as memory consumption. While CPU utilization is important in CPU intensive applications to keep service availability.

When taking scaling decisions we consider 2 types of cost factors.
1 - Resource acquiring cost (C_R)
2 - Losses due to service level violations(C_V)

Here the focus is to minimize the total cost by selecting $C_T(n)$:

$$C_T = C_R + C_V$$

Considering both resource cost and the penalty for performance degradation, we define the total cost in the next T minutes as follows.

$$C_t(n) = C_{ins} \times n + C_{ins} \times n \times f(v_i \times \frac{100}{T})$$

$T = Total\ time\ for\ prediction$
$C_t(n) = Total\ cost\ for\ time\ T$
$C_{ins} = Cost\ for\ an\ instance$
$n = Number\ of\ instances$
$v_i = Violation\ time$

Here f is a predefined function which calculates a penalizing factor given the percentage of performance degradation.

5.1.1 Prediction-Based Auto Scaling Decision Making Procedure

Input :
Predicted workload for T time period with sampling distance of t (Example : Predictions for next 15 minutes with 1 minute intervals, T = 15 , t = 1)
Cost of an instance
Maximum and Minimum number of instances

Process :
Define penalty function f Calculate the number of instances required n based on heuristic

$$n_{opt} = argmin_{n \in N\ \wedge\ n \in [min,max]} C_t(n)$$

Note: Even though we make our decision based on prediction for next T minutes, system will recalculate n_{opt} more frequently (i.e. with time interval less than T) and supersede the decision before.

Output :
Number of instances required for the system (n_{opt})

5.1.2 Intuition Behind the Scaling Algorithm

Algorithm propose to calculate both violation cost and the resource allocation cost at different VM counts. When considered the thicker line in the bottom, violation percentage will be higher while resource cost is lower. Increasing the number of instances increases resource cost while decreasing the violation and the penalty cost.At top most level in

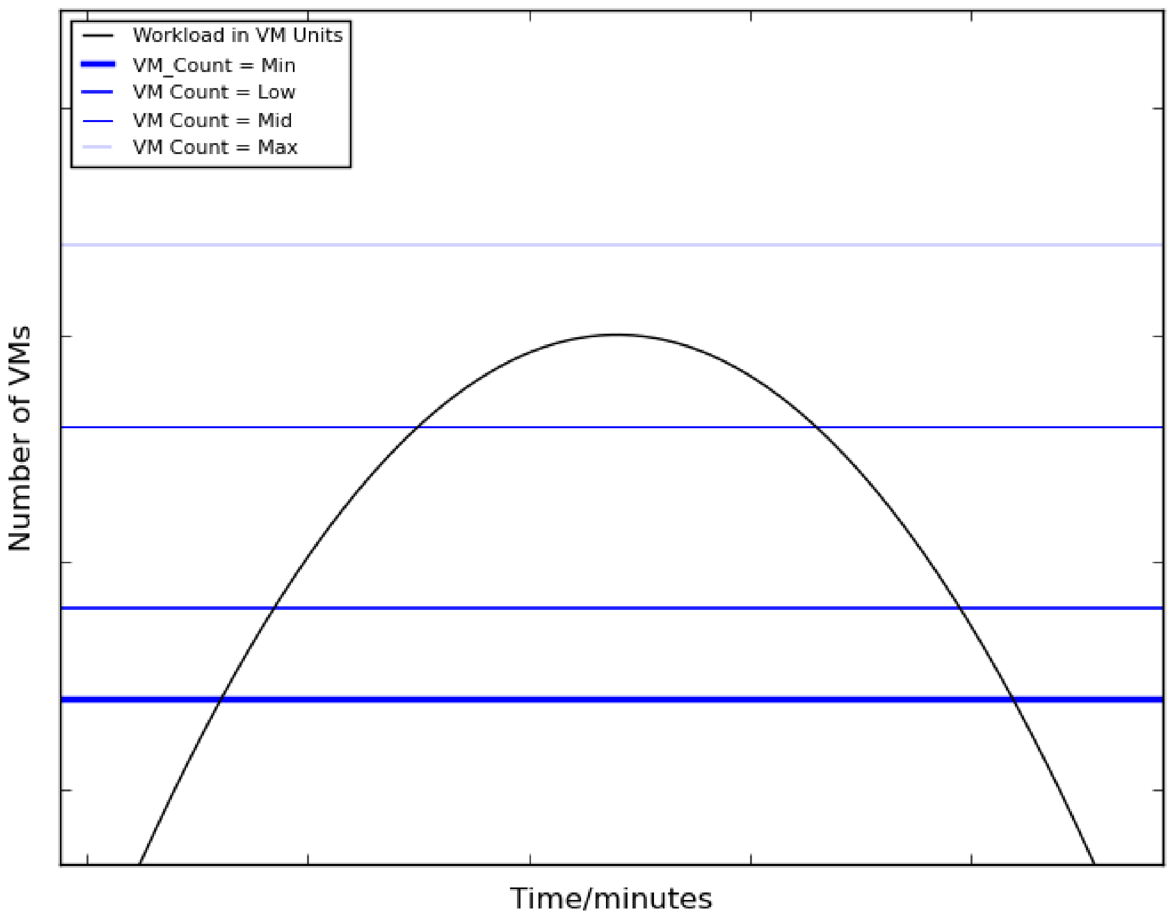


Figure 5.1: Calculation Optimum Number of Instances n_{opt}

5.2 Pricing Model Awareness

Our proposed solution also considers the pricing model of the underlying IaaS layer when taking auto scaling decisions. We adapted the smart killing feature proposed in [[?]] after evaluating the concept. Cloud providers like AWS bill customers on a per-hour basis, which means a user will be charged for one hour even if an instance is used only for a few minutes. Smart killing suggests that an instance should not be spin downed if it has not completed a full billing cycle. Considering practical issues such as the time required to gracefully shut down an instance, we spin down an instance only if it used for 50 to 57 minutes in its billing hour. However, smart killing is only useful for IaaS models with relatively long billing cycles.

5.3 Simulation Setup

5.3.1 Setup and Parameter Details

Stratos Prediction and Scaling

We have simulated Apache Stratos prediction (based on motion equation) and the scaling mechanism which include features such as cool down parameters described in Chapter-2.

Reactive, Threshold based Auto-scaling

In simulation all the reactive approaches are coupled with a 80% threshold. Also simulation is done with the assumption that time to spin up an instance is negligible

Proactive Auto scaling

In general any auto scaling solution based on future workload prediction mechanism and make decision based on the anticipated.In this simulation all the proactive auto scaling are based on the ensemble prediction mechanism proposed in Chapter-5 and the scaling algorithm proposed in Chapter 5.1. Following function is used to calculate the penalty factor

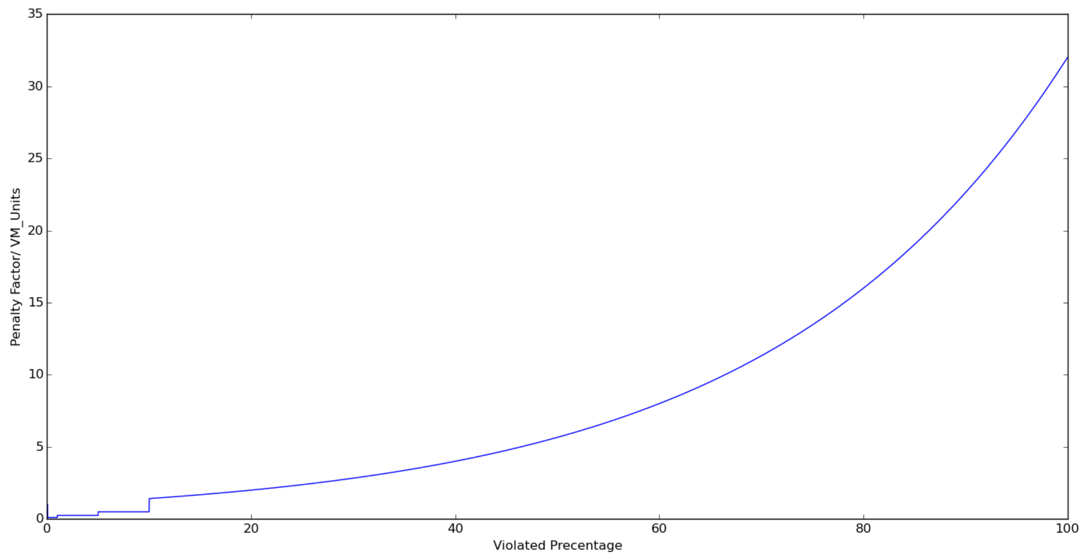


Figure 5.2: Penalty Factor Over Violation Percentage

$$f(x) = \begin{cases} 0 & \text{if } 0 < x \leq 0.05; \\ 0.1 & \text{if } 0.05 < x \leq 1; \\ 0.2 & \text{if } 1 < x \leq 5; \\ 2^{\frac{x}{20}} & \text{if } 5 < x \leq 100; . \end{cases}$$

Blind Killing Vs Smart Killing

In Blind Killing when the system need to be scale down an instance will be killed randomly. Here all the instance are considered equivalent regardless of the current workload or used time.

In smart killing a VM is spin down only if it is used for more than 50 minutes in last billing cycles. Considered the time to gracefully shut down, machine will not be turned off if it is used for more than 57 minutes (Because it will be already billed for next billing cycle when actual spin down happens)
Smart killing is separated from required instance count calculations and can be used with both reactive as well as proactive approaches.

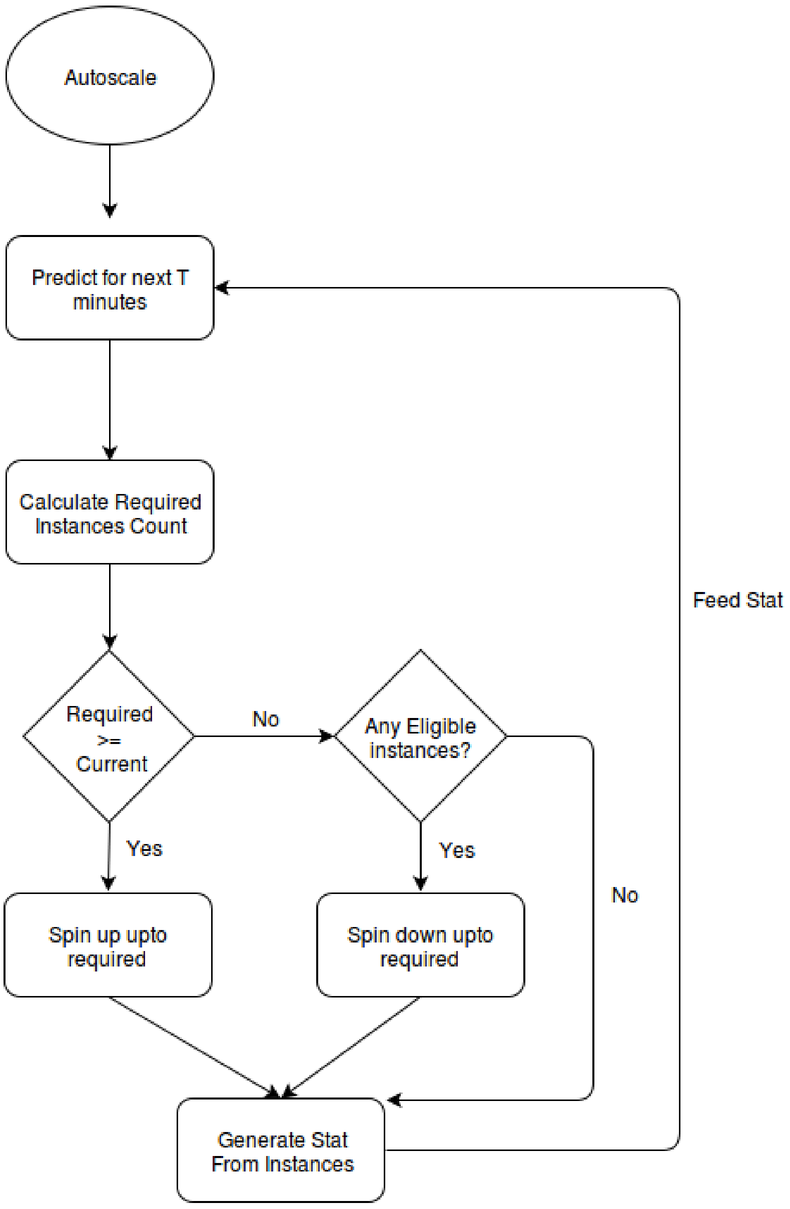


Figure 5.3: Resource Allocation Flowchart

5.4 Simulation Results

5.4.1 Auto scaling Approaches on AWS

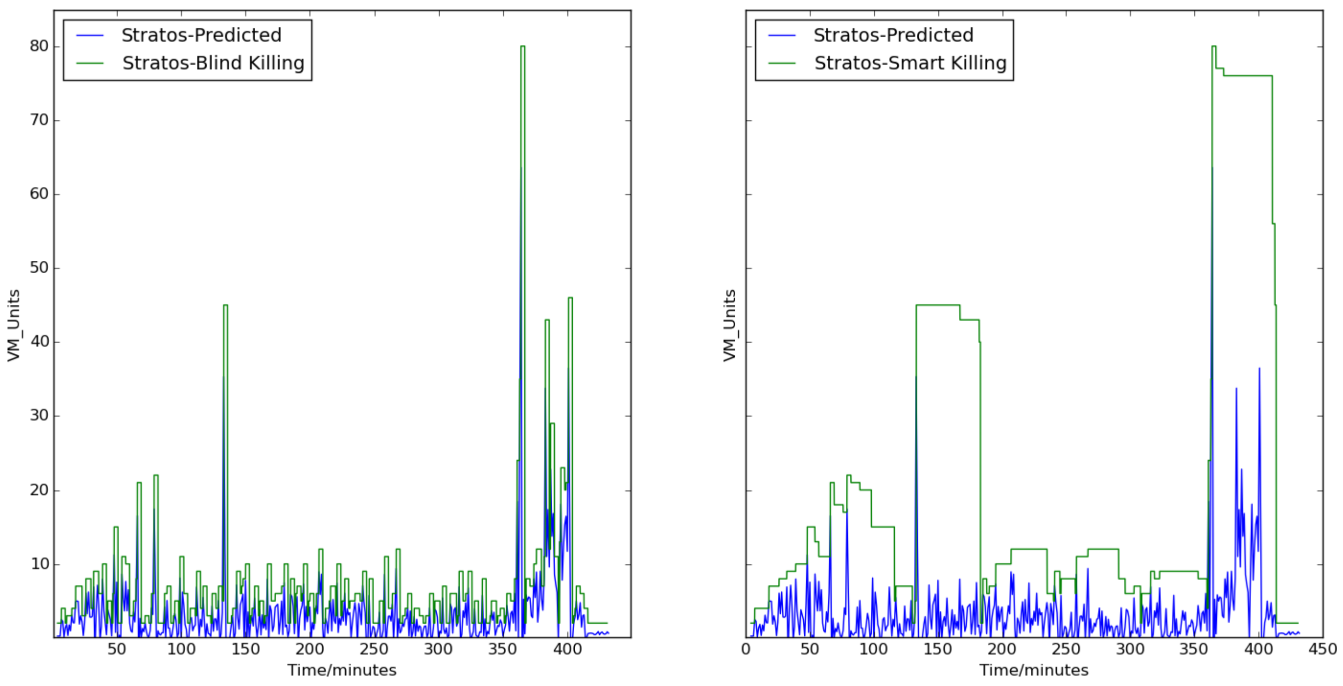


Figure 5.4: Variation of Virtual Machine Count and Load Average Overtime with Stratos Auto-Scaling

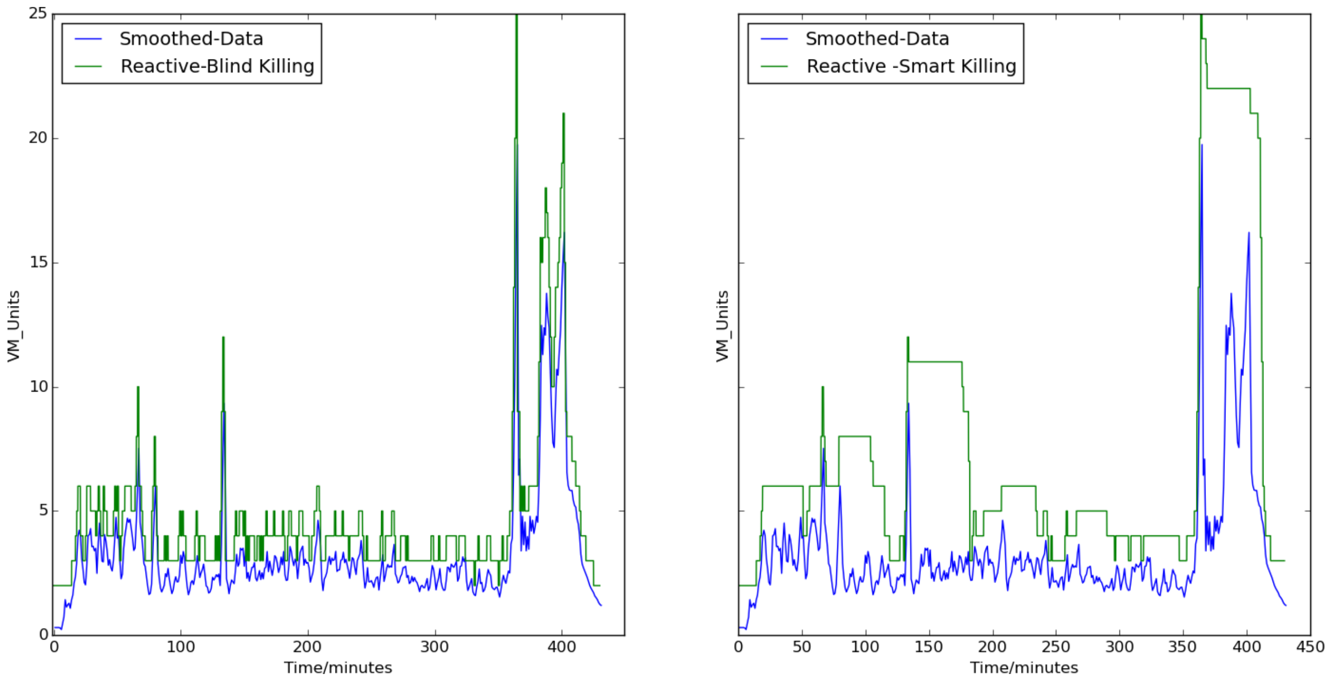
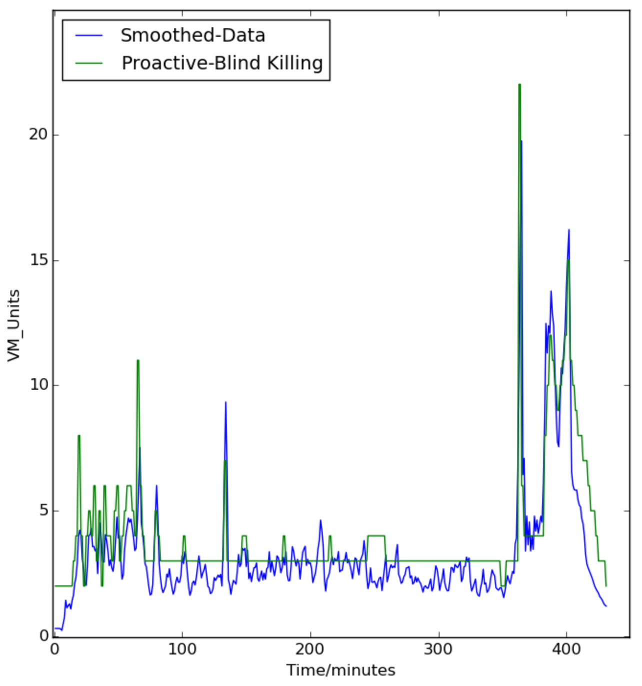


Figure 5.5: Variation of Virtual Machine Count and Load Average Overtime with Reactive Auto-Scaling



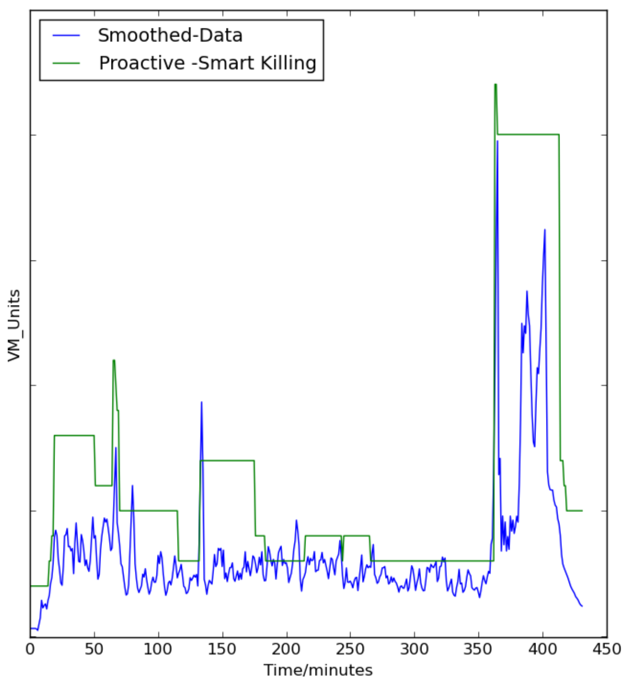


Figure 5.6: Variation of Virtual Machine Count and Load Average Overtime with Proactive Auto-Scaling

Model	Blind Killing	Smart Killing
Stratos Prediction	34.29	12.35
Reactive	9.14	4.13
Proactive	5.53	3.25

Table 5.1: Resource Cost Incurred with Different Auto-scaling Mechanisms (After 7 Hours)

Proactive Smart Killing

Violation Percentage After 7 Hours : 9.26%
Violation Cost After 7 Hours: 0.235

From the scaling graphs in Figure-5.4 to Figure-5.6, Cost Variation Graph in Figure-5.7 and Table-5.1 we conclude that implementing smart killing on auto scaling solutions improve resource utilization while reducing the cost significantly in most cases, regardless of the auto scaling approach (reactive or proactive). We also conclude the proposed proactive scaling approach outperforms the reactive threshold approach, considering QoS as well as resource cost.

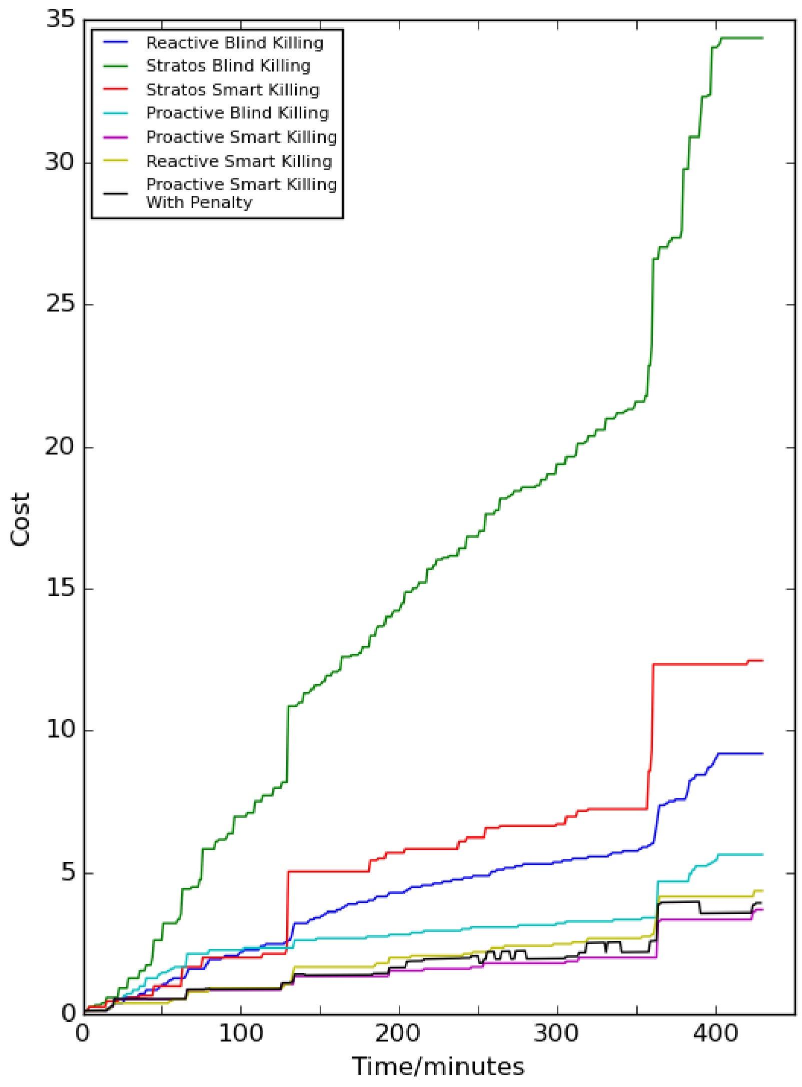


Figure 5.7: Variation of Cost Over time

5.4.2 Evaluation of Scaling Algorithm With Different Instance Sizes

Units(X=M3.Medium)	Resource Cost	Violation Cost	Total Cost	Violation Percentage %
X	**3.25**	0.235	**3.485**	9.26
2X	3.617	0.240	3.857	3.738
4X	5.09	**0.192**	5.282	**0.932**

Table 5.2: Comparison of Costs and Violations with Different Instances

It can be observed that smaller the instance, lower the resource cost and total cost. But percentage of performance violations tends to be higher. On the other hand larger instance incur higher resource cost but lower performance degradation. At top level in the graph performance violation is cost therefore no violation incurred.

5.4.3 Cost Optimizing on IaaS with Short Billing Cycles

Unlike AWS which has minimum billing period of 1 Hour, some other IaaS like GCE has per minute billing with minimum of 10 minutes. (i.e you are billed at least for 10 minutes when spin up an instance, any machine used more than 10 minutes would be calculated for next minute time). Our simulation results as well as practical concerns suggest that smart killing can not be used on those systems. Other optimization techniques has to be implemented considering factors such as monthly discounts, time to spin up an instance on these systems.We simulated cost optimizing mechanism considering GCE. However results do not show a significant gain in terms of cost.

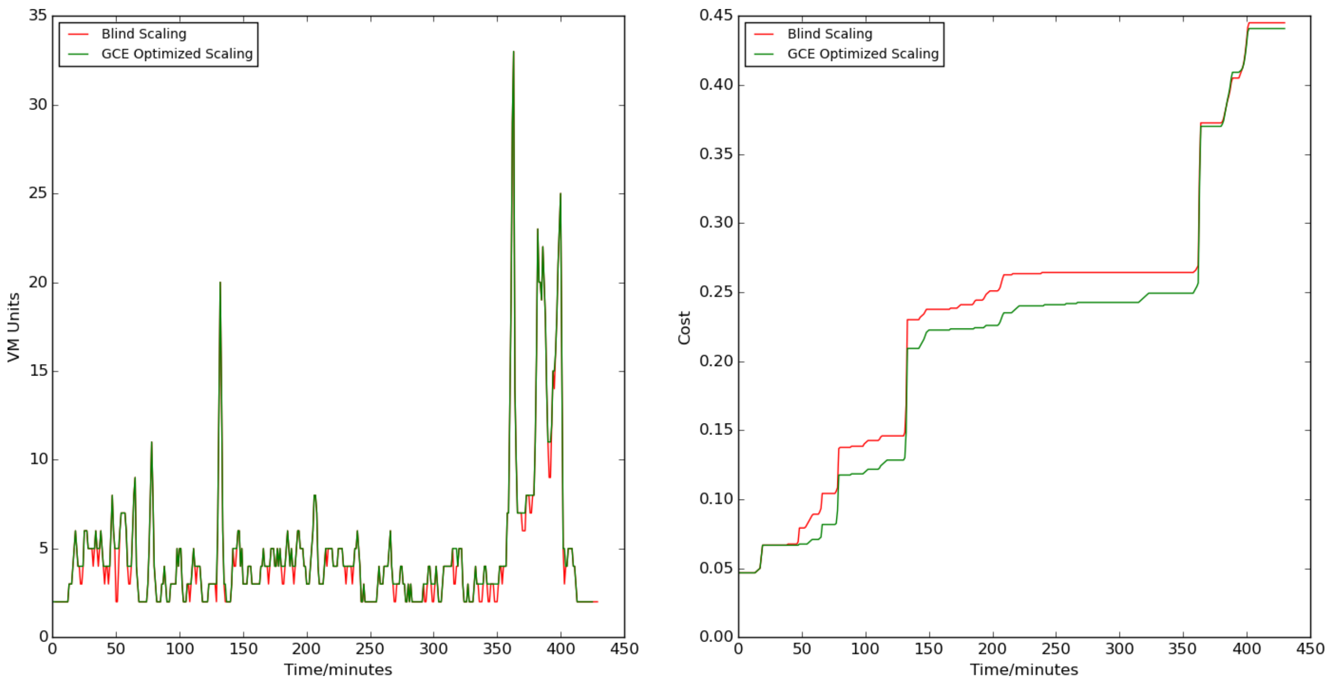


Figure 5.8: Cost Optimization on Google Compute Engine

Chapter 6

inteliScaler Solution

In order to address the issues identified in our performance analysis, we present a complete solution combining our ensemble model for prediction, cost model for cost optimization and smart killing feature for efficient resource allocation. In our solution we mitigate the problem of comprehending a threshold value by forecasting the workload ahead so the auto-scaler can decide at which point the resources should be scaled out or scaled in, looking at the future workload instead of waiting for a trigger. Smart killing feature would apprehend the utilization of each VM instance and scale out instances only when their lease periods are about to expire. It would also save an extra cost by mitigating the requirement to spin up another instance on a sudden fluctuation of the workload within the paid hour.

6.1 Overview

The individual solutions described in Chapter 4 and Chapter 5 solve each aspect of the auto scaling workflow individually. We now combine each solution to provide a better auto scaling service in terms of cost saving, resource provisioning and QoS for the PaaS cloud.

6.1.1 Proactive Auto Scaling

Using our ensemble model we can predict future workload for a window size of 15 minutes with a good accuracy as detailed in Chapter 4. Using Apache Stratos inbuilt architecture, we feed our ensemble model with frequent health stats from each cartridge agent and cluster at a frequency of once in every 15 seconds. Using these values we predict the workload for a 15 minute window, which covers the time for spawning and fault handling, and enables the system to detect spikes ahead of time in order to determine a suitable scaling decision. By using an ensemble model to predict the workload we are able to capture irregular patterns, detect spikes and minimize the resulting errors.

6.1.2 Cost Saving

One major drawback of current PaaS auto scalers is that they neglect the cost factor in their auto scaling solutions. By introducing a cost optimizer where the system would evaluate the cost involved with auto scaling, we have reduced the overall cost drastically compared to the current implementation of Stratos. We consider acquisition cost as well as SLA violation cost in order to complete the equation where both operational cost and service cost is considered.

6.1.3 Efficient Resource Allocation

Blind killing is a common feature among the PaaS auto scalers where the system terminates a virtual machine when the required workload is well below a given threshold, even though payment for the rest of the hour has already been made (applicable for hourly charging schemes).

We introduce a smart killing feature where each machine will be monitored and terminated only if its payment time is about to expire and there is no significant workload predicted for the future. By doing so we were able to eliminate the inefficiency of resource allocation where virtual machines are terminated and new machines are spawned in a small interval period.

6.2 Architectural Design

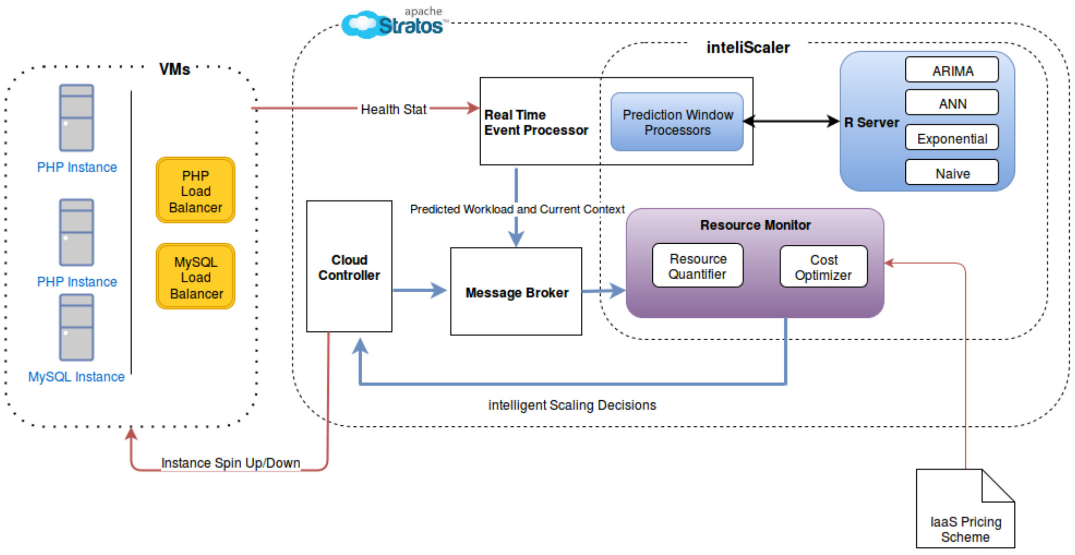


Figure 6.1: System Level Architecture of inteliScaler

The left layer is the IaaS layer where applications, databases and load balancers are deployed on Virtual Machines and communicated using jclouds as shown in figure 2.5. Hence it is capable of catering various IaaS including but not limited to EC2, GCE, OpenStack, CloudStack etc. Stratos use Cloud Controller to communicate to the underlying IaaS and listens to the messages from instances and update the routine topology periodically. By subscribing to the topology published by cloud controller, each load balancer will update their own routing tables.

Apart from the main components in Apache Stratos our solution replaces the autoscaler component while adding additional sub-components into the CEP. In our implementation we have three main components that deals with the auto scaling process.

Prediction Window Processors do real-time monitoring based on the health statistics that are published to the CEP from the cartridges and services. It analyze all the streams that are sent to it and sends the predicted workload collected from the R Server.

RServe is the prediction component where statistics are fed by the window processors are used by four models with a error base ensemble technique to calculate accurate and fast predictions.

Resource Monitor is the cost saving component where the resource quantifier will be calculating the number of instances required based on the prediction and operational cost (acquisition cost and SLA violation cost). Cost optimizer will be performing the scale-up and scale-down actions based on the usage of each resource.

Message Broker handles the communication among all the components in a loosely coupled manner. Currently our solution is deployed on Apache ActiveMQ; however, Apache Stratos supports any Advanced Message Queuing Protocol (AMQP) Message Broker.

6.3 Implementation

6.3.1 Prediction

Goals

To implement the real time forecasting model described in Section 4,we were required to achive several implementation level goals,

1. Forecasting models should be trained real time and provide results within bounded period of time (1 minute in current implementation)
2. Forecasts should be sufficiently accurate for a longer time futue horizon.
3. Amount of the past workload history remain in the memory should be sufficiently large to provide accurate results while small enough to be able to train the models real time.
4. Since the scaling solution mentioned is multifactored(based on CPU,Memory,Request in flight), forecasts should be calculated for each factor parallely.

Possible development options

We looked into several options of implementing our forecasting solution within inteliScaler. To implement our ensemble forecasting solution we required the implementations of low level time series and machine learning models.

SuanShu: SuanShu is an object-oriented, high performance, extensively tested, and professionally documented math library. It is a large collection of numerical algorithms so coded such that they are solidly object-oriented.

The Apache Commons Mathematics Library: Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang.

forecasting package: Methods and tools for displaying and analysing univariate time series forecasts including exponential smoothing via state space models and automatic ARIMA modelling.

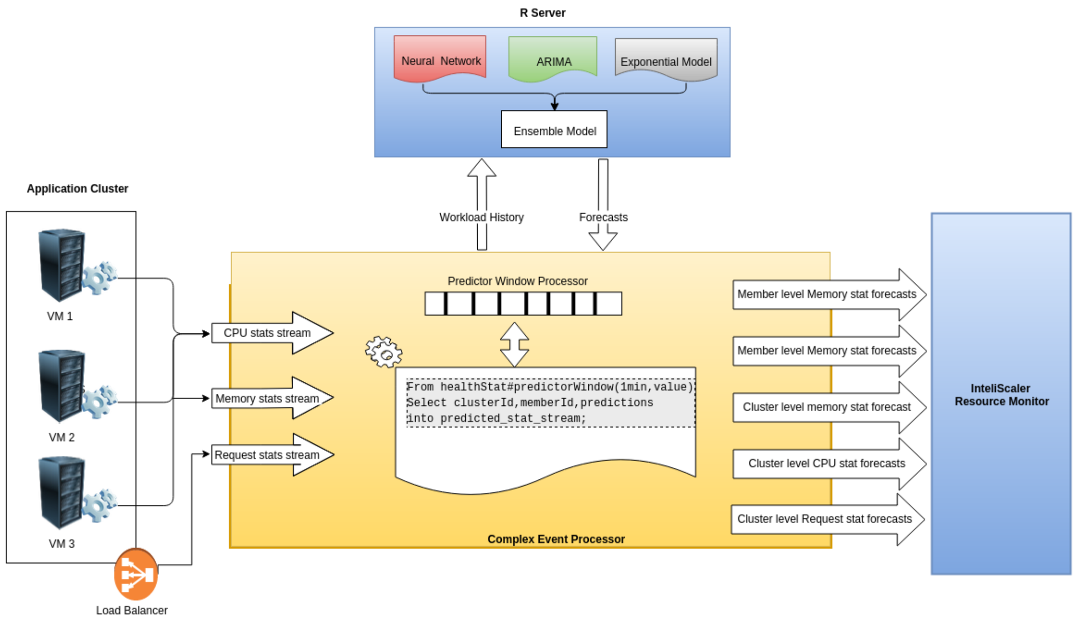
Sunshu is a powerful computation library, but it is a paid library. Since Apache Stratos is product we did not integrate commercial products like Sunshu.

Apache common math is a widely used open source library with JAVA support. Even though this supports statistical time series models like ARIMA, Exponential model, this library doesn't not have automatic parameter estimation by minimizing forecasting error.
R forecast package which was used in Section 4, has excellent functionalities like automatic model selection and parameter estimation for various statistical and machine learning models for time series analysis. This library is heavily uses in researh domain for time series forecasting. Since our overall implementation is based on Java, we required to execute model implemented in R through Java. For that we found several options

1. JRI : RI is a Java/R Interface, which allows to run R inside Java applications as a single thread. Basically it loads R dynamic library into Java and provides a Java API to R functionality. It supports both simple calls to R functions and a full running R scripts.
2. JPMML : By converting forcasting model into PMML defintion file and generated the predictive model in Java.Then the model can be evaluated using native java methods.

3. Rserve :Rserve is a TCP/IP server which allows other programs to use facilities of R from various languages without the need to initialize R or link against R library. Every connection has a separate workspace and working directory. Typical use is to integrate R backend for computation of statstical models, plots etc. in other applications.

Even though JPMML support various machine learing models, it doesn't support times series models in R- forecast package. We first tried JRI interface to execute R- forecasting model through JAVA. But JRI has several restrictions like, not being thread seaf and unability to instantiate multiple R-engines within single JVM. So that we integrated the predictive model with Stratos using Rserve conection.


R Server
Neural Network
ARIMA
Exponential Model
Ensemble Model
Application Cluster
Workload History
Forecasts
VM 1
VM 2
VM 3
Load Balancer
CPU stats stream
Memory stats stream
Request stats stream
Predictor Window Processor
Member level Memory stat forecasts
Member level Memory stat forecasts
Cluster level memory stat forecast
Cluster level CPU stat forecasts
Cluster level Request stat forecasts
Complex Event Processor
InteliScaler Resource Monitor

```
From healthStat#predictorWindow(1min,value)
Select clusterId,memberId,predictions
into predicted_stat_stream;
```

Figure 6.2: High level Architecture of Prediction component

Load average and Memory consumption stats are published to the CEP via application cartridge instances withing the virtual machine in application cluster. Request In flight stream will be published by load balancer of the cluster to CEP in 15 second intervals. CEP queries are written to process each stream and produces the forecasts for next n number of minutes. The input event streams(health stats, request in flight) comes in 15 seconds interval while we need to output forecast for next n minutes as output stream in 1 minute interval. Hence the health-stat events within a minute should be accumulated and processed in each minute. So that window processor should have been implement for our use case.

Usually there are two types of window processors in complex event processing engines.

- Batch window processor which accumulate the events within a given period and processed after the period is completed.
- Sliding Window Processor, which accumulate the events within fixed interval and processed the event window in each new event arrival while slide trough the history of events.

In our real-time prediction solution, we required a customized version of sliding and batch window processed to generate forecast for each stream.

Hear we keep track 2 types of lists for processing events called new event list and global window list. New event list contains the events came in last 1 minutes. By using the accumulated list, the average value over last minute is calculated. Then average value is put into the global event list which contains the averaged values for each minute. This global workload event list is taken as the input for the ensemble model.

Implementation of the predictive model

As our research work suggests (Section 3), an ensemble solution which can adapt to the different workload patterns is the desirable option for a PaaS platform. We chose four time series forecasting models as the base models of our ensemble approach, namely naive forecast, ARIMA, exponential model and a neural network. Details of the implementations of these models are already mentioned in Section 4.1.3. We used the weighting mechanism described in Section 4.2.1 to weigh the results from individual models to generate the ensemble output.

As an optimization step, only a single R server connection is created for each active event stream (during the first processing iteration of the stream), which is then persisted for serving further computation requests in subsequent iterations, thereby avoiding the processing overhead of creating separate R server connections for every iteration.

Special Concerns

Amount of data points stored: Machine learning models like neural networks require significant amounts of data points for producing accurate forecasts. However, since our scenario involves online training, the size of the dataset that can be used for training is bounded by the limited amount of time available before the next computing cycle. We have experimentally determined that around 3000 data records can be safely trained for generating results within a bound of approximately 20 seconds.

Memory usage: Even though it is more desirable to perform predictions using raw data points received directly from health stat and RIF publishers at 15-seconds intervals, storing such a large amount of data in memory has scalability concerns. Instead, we store and utilize average values of data points calculated over 1-minute time intervals in the prediction process.

Handling out-of-range forecasts and outliers: During our preliminary testing, we noticed that the neural network model produces extreme deviations which cannot be readily compensated by the ensemble process, particularly during early stages of online prediction where only a limited amount of data points is available for training. In order to mitigate this issue, we used outlier analysis techniques to readily capture such drastic deviations and discard them before feeding the individual results to the ensemble model.

6.3.2 Resource Quantifier

For each member in every cluster, prediction window processors publishes a workload forecasts a predictions per minute for a 15 minute window. These values are used by the resource quantifier to interpolate the curve in which best suit the provided prediction points. Using the SplineInterpolator class in Apache commons math3 package, we compute a natural cubic spline interpolation for the dataset of each member. The curve is fitted using several cubic polynomials defined over the sub-intervals in the 15 minutes window, known as 'knot points'. Using such a method against some of the popular methods like Lagrange interpolation is beneficial as the curve might be of order N and lagrange interpolation displays a unstable oscillation property for higher order polynomials, where as using cubic spline interpolation is more accurate and captures the exact pattern for any polynomial.

The fitted forecast for each member is then averaged to calculate the average cluster requirement for each metric (except for request in flight metric) for the 15 minute window. In order to calculate the optimum number of instance required to handle the workload, we enumerate from the minimum allowed instance count to the maximum and check the percentage of violation of SLA to determine the operational cost for each configuration. If there is a minimum cost with no violation then that configuration will be selected or if the maximum configuration has the least violation percentage it will be the configuration that will be selected. The selected optimum configuration is then forwarded to the cost optimizer where it will look into the resources usage and decide on a suitable scaling action.

6.3.3 Smart killing

Since the main concern of smart killing is the avoidance of premature termination of VMs for which a billing cycle has already been started (rather than interfering with the instance start-up process), it makes more sense to include smart killing logic in the scale down portion of the auto-scaling algorithm. Accordingly, we have updated the section of the JBoss Drools-based rule engine to incorporate smart killing.

Approach

Our approach assumes a 60-minute billing cycle time, consistent with the current specifications of AWS. Since it would be pointless to terminate an instance before the lapse of a sufficiently large fraction of its billing cycle, we ignore all instances that have not completed at least 50 minutes of their current billing cycle, in selecting termination candidates during a scale down. While these time values are specific to AWS, they can be easily adjusted to fit any other IaaS provider with sufficiently long billing cycles.

As different instances of the cluster may be subjected to different load levels at the time of the termination request, they may require different time durations for a graceful shutdown. This poses a challenge where we have to ensure that the instance does not move into the next billing cycle while it is terminating (because most of the commercial IaaS providers like AWS consider the termination completion time as the end of the running time of the instance), while also allowing the instance to continue its normal operations as far (towards the end of the billing cycle) as possible.

Stratos auto-scaler currently addresses part of this issue by selecting low-load members for quick termination based on a custom parameter *overall load* (a combination of load average and memory consumption), while allowing the high-load members to continue handling their current load. However, in our case, we cannot solely depend on the load levels of instances as we also need to consider the remaining times (before the next billing cycle) for each instance. Currently we address this problem by first selecting all possible termination candidates and then partitioning them as low-load (below 50% of overall load) and high-load. We do not consider a high-load instance as a suitable candidate unless it has at least 5 minutes of its billing cycle remaining, whereas the remaining time requirement is relaxed to 3 minutes for low-load instances.

Algorithm

The following algorithm is executed when a scale down is requested by the workload prediction logic:

- The rule engine ensures that the number of non-terminated instances is above the minimum permitted value for the cluster.
- The request is passed through a cool down logic. Similar to the existing implementation in the Stratos auto-scaler, this logic prevents the propagation of the request unless three consecutive scale down requests already been received. This helps mitigate situations of fluctuating workload patterns, and the possibility of thrashing in case of variations in marginally handled workloads.
- n, the number of instances to be terminated, is determined, subject to the cluster level minimum instance count constraint.
- The rule engine iterates over the available members (instances) in the cluster, looking for suitable termination candidates based on overall load and remaining billing cycle time. A member is deemed suitable if it meets the minimum uptime as well as the load-uptime combination constraints described above.
- Chosen candidates are sorted in the increasing order of their remaining cycle time.
- The first (oldest) n members from the pool are chosen and terminated.

Special Concerns

A major issue of this smart killing approach is that the selection of boundary values for high and low overall load, as well as the time durations allocated for the termination of high-load and low-load members, need to be determined experimentally. Ideal values for these parameters could be derived through analysis of past smart killing statistics.

6.4 Results

We ran several workloads against Apache Stratos and inteliScaler for evaluation of their performances. Both PaaS were deployed on AWS EC2 system as described in figure 3.8. Few workload patterns with significant differences are described below.

6.4.1 Test on Mock IaaS

We also tested few workloads on Mock IaaS to simulate the performances of Apache Stratos and inteliScaler. For the purpose of testing we set the billing cycle to 10 minutes.

As shown in figure 6.3 both Stratos configuration has a low resource allocation to handle the given workload but comparatively Low threshold configuration has a higher allocation. For the same workload inteliScaler has a higher resource allocation as it tries to give 100% QoS for the given workload. In terms of QoS, inteliScaler outperforms both configuration of Stratos and has sufficient resources allocated ahead of time. Since this is simulation, we simply cannot conclude which is better to handle this workload. But we will be testing the same workload on AWS EC2 setup on the next section to get an clear idea.

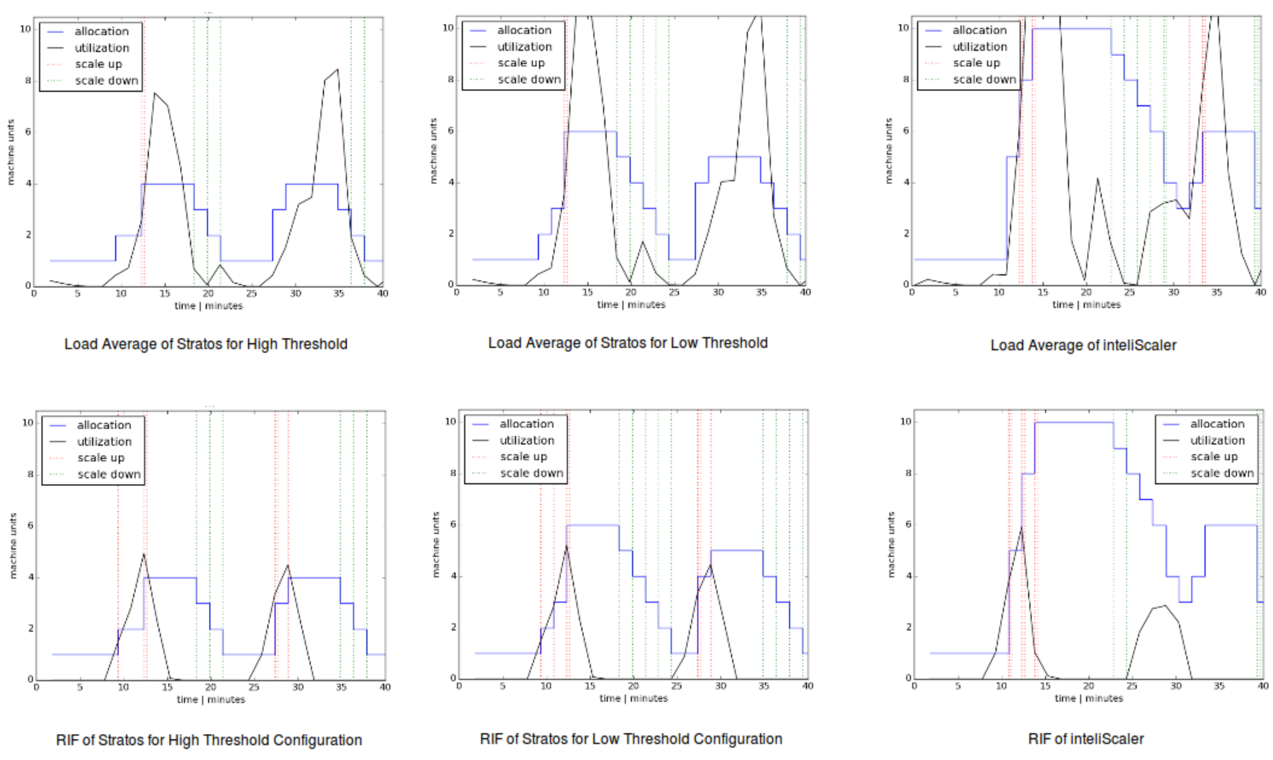


Figure 6.3: Performance comparison in Mock IaaS Setup

6.4.2 Test on AWS EC2

We ran several workloads, with drastically different characteristics, on the actual implementation of Apache Stratos and inteliScaler deployed in AWS EC2 setup as described in 3.8

Workload I

First we tested the same workload that was used earlier as shown in table 3.2 against inteliScaler. Configurations of the Stratos are same as described in table 3.3 and table 3.4.

Table 6.1: QoS summary for evaluation for Workload I on AWS EC2

test case	average response time	requests initiated	requests completed	time out errors
low threshold	6.3839 s	254535	203067	2511
high threshold	4.4954 s	280477	191891	11863
inteliScaler	5.1288 s	286180	261979	2278

It is clear that inteliScaler outperforms both configurations of Stratos as shown in figure 6.4. InteliScaler has allocated less number of VM's to handle the same workload. In terms of QoS inteliScaler has a success ratio of 91.54%, which is significantly higher than, the 80% of low threshold configuration and 68% of high threshold configuration. Number of time out errors is also significantly low where in inteliScaler it is 0.8%, in low threshold configuration it is 0.9% and in high threshold configuration it is 4.2%.

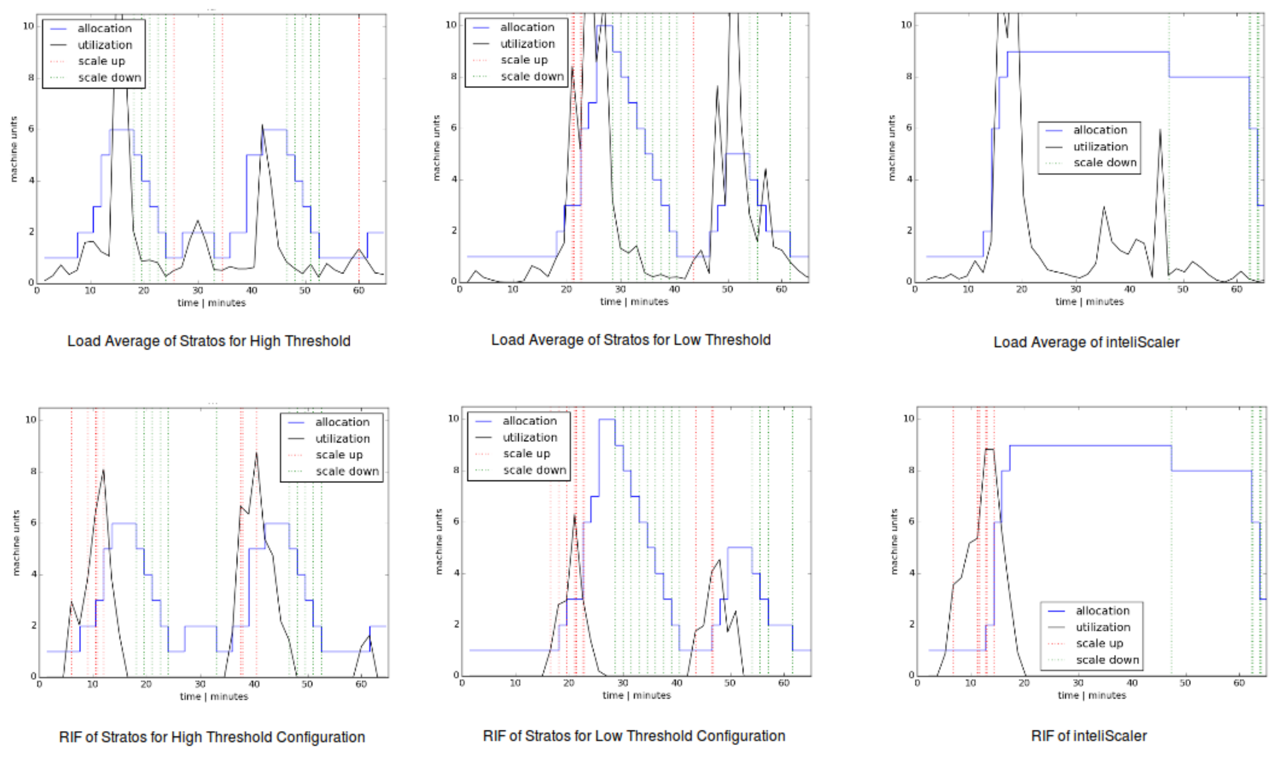


Figure 6.4: Performance comparison in AWS EC2 Setup for Workload I

Workload II

We also tested both Stratos and inteliScaler with a fluctuating yet growing workload as described in table 6.2.

Table 6.2: Workload II for AWS Setup

Segment	1	2	3	4	5	6	7	8	9	10
Duration (seconds)	90	90	90	90	90	90	90	90	90	90
Users	200	100	400	200	600	400	800	600	1000	800
Transition (seconds)	30	30	30	30	30	30	30	30	30	30

Table 6.3: QoS summary for evaluation of Workload II on AWS EC2

test case	average response time	requests initiated	requests completed	time out errors
Stratos	2.7150 s	178848	102671	16257
inteliScaler	1.1207 s	214041	201719	6988

As shown in figure 6.5 inteliScaler has the lowest resource allocation of 10 VMs whereas for the same workload Apache Stratos allocates 15 VMs, saving the cost for 5 VM's. Also the QoS of inteliScaler is better than Stratos as shown in table 6.3. inteliScaler has responded to 94.24% of the requests generated whereas Stratos has responded to only for 57.41% of the total requests generated. There is a significant time out and drop off error in Stratos compared to inteliScaler, which is the reason for these figures.

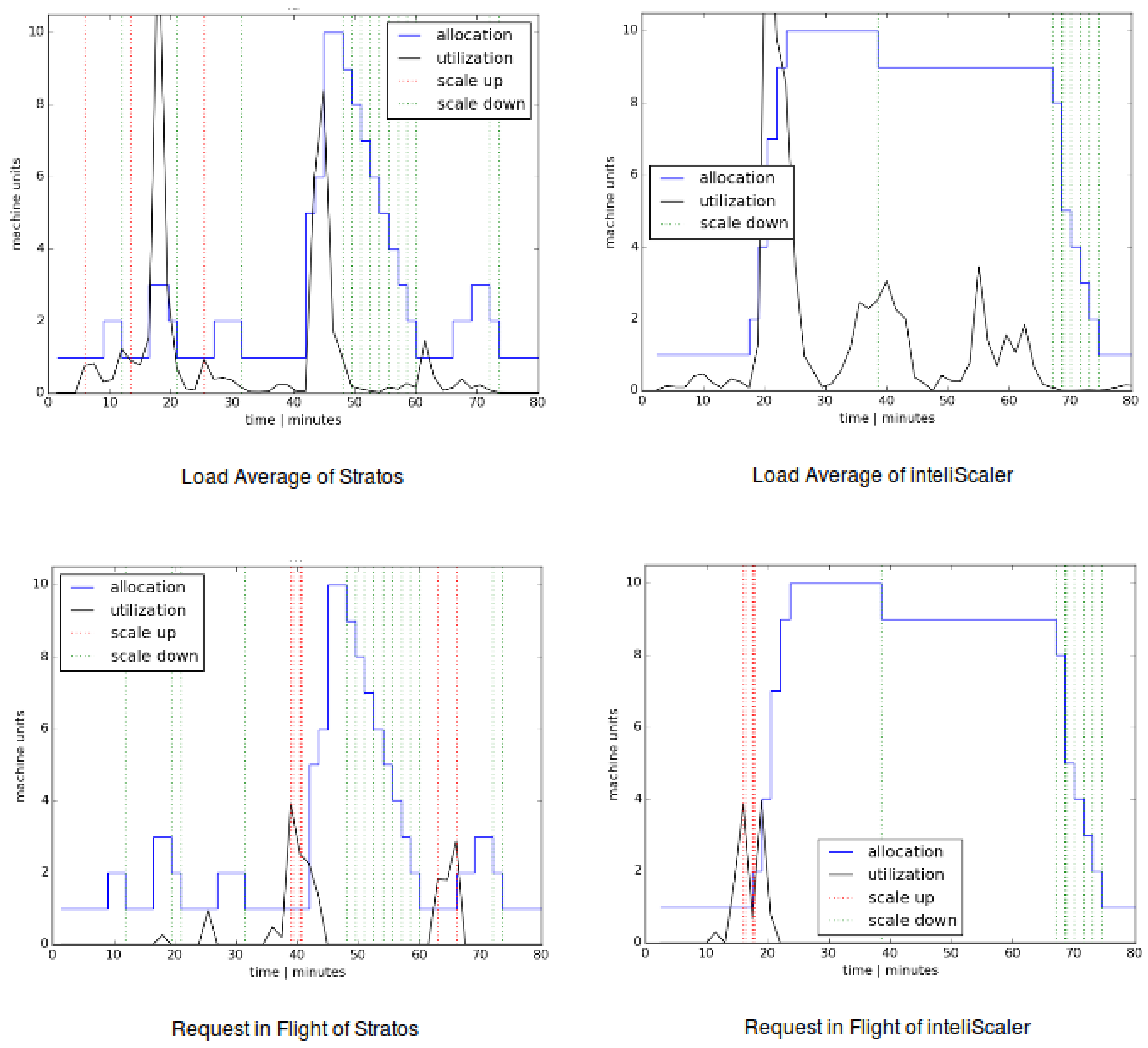


Figure 6.5: Performance comparison in AWS EC2 Setup for Workload II

Chapter 7

Summary

Auto-scaling is the process of dynamically allocating and deallocating resources for particular application deployed in the cloud. This is of vital importance for clients to optimize their resource utilization. The goal of cloud auto-scaling mechanisms (auto-scaler) is to achieve higher levels of Quality of Service while minimize the associated costs.

We have identified two major challenges faced by cloud auto-scaling systems. Firstly, an auto-scaler need to be aware of the workload that the system has to deal with. Secondly, an auto-scaler should allocate the right amount of resources to the system in a cost effective manner while preserving Quality of Service.

There are two main approaches in cloud computing domain to address the first challenge. Depending on autoscaling approach, auto-scaler would gain workload awareness in proactive or reactive manner. In reactive approach autoscaling decision would be triggered by predefined set of events. In contrast proactive autoscaling mechanisms forecast the workload ahead so the auto-scaler based can make decisions based on anticipated workload instead of waiting for a trigger. However major challenge in cloud workload prediction is to come up with solution that would perform with different workload patterns. For example a model that perform well on workloads with seasonal trends will not perform well with frequently fluctuating loads.

We opted proactive auto-scaling over reactive in our solution because proactive mechanism supported by accurate prediction mechanism enable auto-scaler to make more detailed decisions.Reactive approach used in most PaaS auto-scaler in our view is much simpler, but greatly reduce the solution space available when addressing the second (resource allocation) problem. In Chapter 4 we proposed workload prediction mechanism based on time series forecasting as well as machine learning techniques. Results has shown that ensemble method proposed outperform individual techniques in term of accuracy.

Significant amount of research has been conducted (discussed in Chapter 2) in relation to resource allocation problem as well. But almost all the available PaaS solutions are built on rule based scaling. For example in rule driven approach spin up decision will be taken when the average memory consumption of the cluster of virtual machines is over 75%. These rule based mechanisms mostly rely on user defined threshold parameters. One of the major drawback in this rule based threshold-driven auto-scaling is that the user is expected to be a domain expert and should be capable of setting up threshold values such as memory usage and CPU consumption for his application in order for configuring policies or rules to govern the scaling decisions. On the other hand, mapping application metrics such as response time and transactions per unit time to system level metrics such as CPU usage and disk IO rates is a research level problem, which an average user cannot comprehend
In Chapter 5 we proposed a greedy heuristic scaling algorithm considering both QoS as well

as cost factors. In the algorithm we introduced penalty factor for performance degradation, an idea inspired by penalties introduced in SLA of popular PaaS such as Google App Engine. The scaling algorithm evaluates all possibilities and selects the optimum resource configuration considering the lease cost and penalty due to performance degradation. This scaling mechanism mitigates the problem of comprehending a threshold values in rule based scaling.

Other than introducing a scaling algorithm to calculate required resources, we take a novel perspective in addressing resource allocation problem injecting pricing model awareness to our auto-scaling solution as we see ignorance of the pricing model in scaling decision making is major drawback in PaaS auto-scalers available. For a motivation example let us consider a typical application deployed on top of Amazon Web Service (AWS) PaaS. On a sudden fluctuation in the workload, a typical auto-scaler would scale-out to spin up a new VM instance and when the workload is back to normal the auto-scaler would scale-in by blindly killing one of the VM instances which has already been paid for for an hour. Thus the customer has lost a 50 minutes utilization of the instance. Smart Killing feature we adapted to our solution from [Appscale] would apprehend the utilization of each VM instance and scale out instances only when their lease periods are about to expire. It would also save an extra cost by mitigating the requirement to spin up another instance on a sudden fluctuation of the workload within the paid hour.

In an attempt to implement our solution and demonstrate the effects of a PaaS auto-scaler, we target Apache Stratos PaaS (discussed in Chapter 6); however, the techniques detailed here are extensible to other PaaS systems as well. Apache Stratos, an open source PaaS framework offering multi-tenancy and multi-cloud deployment capabilities by encapsulating the details to the level of reduced granularity and complexity of a PaaS. Stratos supports multiple IaaS providers, including AWS, OpenStack, GCE (Google Compute Engine). Stratos auto-scaler is based on policy-driven decisions, which performs workload prediction for a small time window (usually a few minutes) but does not utilize resource optimization approaches explicitly, thereby incurring unnecessary costs to the customer, such as over-provisioning of resources and naive scale down decisions. Apart from the typical characteristics of a PaaS auto-scaler, Stratos offer other features such as a modular architecture allowing easy modifications, and the availability of a mock IaaS as a component for testing and evaluation, which would greatly assist any research in terms of monetary cost and implementation cost.

We evaluate inteliScaler by deploying Apache Stratos on AWS Elastic Cloud Computing(EC2). We deploy the three-tier bidding benchmark RUBiS as the user application and experiment with various workload traces. Our results demonstrate that inteliScaler successfully scales the application in response to fluctuating workloads, without user intervention and without offline profiling. More importantly, we compare our solution with existing rule based triggers and show that inteliScaler is superior to such approaches. Auto scaling is of vital importance to realize the full potential of cloud computing. However, current auto-scaling solutions available in PaaS cloud remain primitive in several respects.

7.1 Problems and Challenges

During the work of our research we had many challenges to overcome since our research is based on cloud technology. One of the major challenge was that setting up a cloud infrastructure for the testing purposes and purchasing readily available cloud services was quite expensive. We applied for a research grant at Amazon Web Services to obtain free access to their cloud services, and we were granted 500 AWS credits for one year period, without unlimited access to all their services. Apart from the research we also obtained 750 machine hours per month for their free tier services which included few EC2 instance types. Through this grant we were able to save a significant amount of cost in acquiring the necessary infrastructure to build our custom test beds, and it also saved us from the time and trouble of setting up our own cloud infrastructure.

Collecting workload data for our simulations was not a trivial task, as there were not much of openly available datasets that could be used in our context. Even though there were some datasets which were popularly cited in the literature, obtaining such datasets and adapting them to our requirements was not an easy task. Gratitude to our supervisor Dr. Dilum we were able to obtain access to several server nodes from the department to do our simulations and generate workloads for our research purpose. Using the resources granted we were able to generate synthetic workloads as well as emulate log traces to produce empirical workloads.

Apart from the infrastructure and resource problems, we encountered several challenges in setting up Apache Stratos PaaS cloud on AWS EC2. Since we did not have much experience in cloud technology, especially in setting up a PaaS in the cloud, we had to troubleshoot many challenges that are fairly common in a distributed system. Through consultation with our external supervisors from WSO2 and the usage of the Apache Stratos developers' mailing list, we were able to navigate through our problems and successfully deploy Apache Stratos as a distributed setup on AWS EC2.

7.2 Future Work

7.2.1 Define Different Penalty Function Based on Expected QoS

It is possible to improve the proposed heuristic by introducing different penalty functions based on the level of service expected by different users. For example, an application (deployed on a PaaS) that supports free as well as paid versions would require different service levels. Our proposed solution can be adapted for such scenarios by defining different penalty functions based on the type of subscription.

7.2.2 Bidding for Spot Instances on AWS

Our research work on cost modeling was focused on on-demand instances. However, IaaS providers such as AWS support other instance types with different payment schemes, such as spot and reserved instances. In spot instances, the IaaS user (PaaS provider in our case) can bid for idling instances. If the bid is higher than the market value, the instance will be assigned to the bidder. An efficient implementation for acquiring spot instances for handling dynamic workload requirements would significantly reduce overall cost.

7.2.3 Inclusion of a Performance Model

Current implementation of inteliScaler is focused on a limited set of instances, since we were not able to conduct research for all the instance types available at AWS due to the cost factor. For completion of the solution, a performance model is required for mapping the workload requirement with a suitable instance type configuration available at the IaaS layer. This depends heavily on the application being deployed, but in general we will be able to accurately define the limitations of each resource type for a generic workload.

7.2.4 Support for Heterogeneity

Our solution as well as the current auto-scaling implementation on Apache Stratos assumes homogeneity of the worker nodes (members or instances). Although this results in a simpler resource management model, it prevents Stratos from enjoying many benefits and profitable aspects of heterogeneous deployment, including spawning of instances with specialized resource capacities (e.g., a memory-optimized instance when the cluster is facing memory deficiencies rather than high load averages or requests-in-flight counts) and choosing combinations of different instance types to fulfill a given resource demand while minimizing total cost. However, implementing heterogeneity on the current Stratos architecture would include a substantial amount of changes to the existing code base.

7.3 Conclusion

Inteliscaler is an attempt to inject intelligence to PaaS auto-scaling domain which mostly rely on primitive scaling mechanisms.Lack of workload awareness being a major issue in current PaaS solutions, we have shown that good workload forecasting mechanism would help PaaS auto-scalers to make more educated decision. Also user- defined, threshold-based scaling mechanisms used in current auto-scalers are not capable of find good balance between in the tradeoff between QoS and the resource cost. Inteliscaler shows the importance of QoS and cost aware scaling mechanism to find right balance in the tradeoff. Further,inteliscaler shows that awareness of prcing model of underlying IaaS layer can help towards significant saving for PaaS users.

Inteliscaler consist of two major components. Workload prediction mechanism inject future workload awareness for decision making. Resource allocation monitoring component is targeted towards cost and QoS aware decision making based on anticipated workload. We implemented proposed solution on Apache Stratos PaaS framework to demonstrate the effectiveness of our solution.

Bibliography

[1] P. M. Mell and T. Grance, “The NIST definition of cloud computing,” tech. rep., 2011.

[2] “Why Apache Stratos is the preferred choice in the PaaS space.” Available at `http://stratos.apache.org/about/why-apache-stratos.html` (2015/07/20).

[3] “Amazon Web Services (AWS) - Cloud Computing Services.” Available at `http://aws.amazon.com` (2015/07/20).

[4] N. Roy, A. Dubey, and A. Gokhale, “Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting,” in *2011 IEEE 4th International Conference on Cloud Computing*, Institute of Electrical & Electronics Engineers (IEEE), jul 2011.

[5] M. A. et al., “A view of cloud computing,” *Communications of the ACM*, vol. 53, p. 50, Apr. 2010.

[6] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” *J Grid Computing*, vol. 12, pp. 559–592, Oct. 2014.

[7] H. Alipour, Y. Liu, and A. Hamou-Lhadj, “Analyzing auto-scaling issues in cloud environments,” in *24th Annual Intl. Conf. on Computer Science and Software Engineering*, CASCON ’14, (Riverton, NJ, USA), pp. 75–89, IBM Corp., 2014.

[8] C. Bunch, V. Arora, N. Chohan, C. Krintz, S. Hegde, and A. Srivastava, “A pluggable autoscaling service for open cloud PaaS systems,” in *5th IEEE Intl. Conf. on Utility and Cloud Computing*, Nov. 2012.

[9] “How PaaS Auto Scaling Works on OpenShift.” Available at `https://blog.openshift.com/how-paas-auto-scaling-works-on-openshift` (2015/07/20).

[10] “What is Cloud Computing? — Basic Concepts and Terminology.” Available at `http://whatiscloud.com/basic_concepts_and_terminology/scaling` (2015/07/20).

[11] J. Yang, C. Liu, Y. Shang, Z. Mao, and J. Chen, “Workload predicting-based automatic scaling in service clouds,” in *6th IEEE Intl. Conf. on Cloud Computing*, June 2013.

[12] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, “Optimal Autoscaling in a IaaS Cloud,” in *9th Intl. Conf. on Autonomic Computing*, Sep 2012.

[13] “Scryer: Netflixs Predictive Auto Scaling Engine.” Available at `http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html` (2015/07/20).

[14] “Application Scaling — OpenShift Developers.” Available at `https://developers.openshift.com/en/managing-scaling.html` (2015/07/20).

[15] “Getting started with the Auto-Scaling service.” Available at `https://www.ng.bluemix.net/docs/services/Auto-Scaling/index.html` (2015/07/20).

[16] M. Mao and M. Humphrey, “Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows,” in *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011.

[17] R. A. Hyndman and G. Athanasopoulos, *Forecasting: Principles and practice.* OTexts, 2013.

[18] “AutoscaleAnalyser.” `https://github.com/hsbhathiya/AutoscaleAnalyser/tree/master/datasets/cloud_traces`, 2015.

[19] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, “Automated Control in Cloud Computing: Challenges and Opportunities,” in *1st workshop on Automated Control for Datacenters and Clouds (ACDC)*, 2009.

[20] H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi, and L. Yuan, “Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers,” in *IEEE Intl. Conf. on Services Computing*, July 2010.

[21] J. Kupferman, J. Silverman, P. Jara, and J. Browne, “Scaling into the cloud.” Available at `http://cs.ucsb.edu/~jkupferman/docs/ScalingIntoTheClouds.pdf` (2015/07/20), 2009.

[22] S. Khatua, A. Ghosh, and N. Mukherjee, “Optimizing the utilization of virtual resources in cloud environment,” in *IEEE Intl. Conf. on Virtual Environments, Human-Computer Interfaces and Measurement Systems*, Sep. 2010.

[23] Z. Gong, X. Gu, and J. Wilkes, “PRESS: PRedictive Elastic ReSource Scaling for cloud systems,” in *Intl. Conf. on Network and Service Management*, Oct. 2010.

[24] S. Islam, J. Keung, K. Lee, and A. Liu, “Empirical prediction models for adaptive resource provisioning in the cloud,” *Future Generation Computer Systems*, vol. 28, pp. 155–162, Jan. 2012.

[25] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, “Adaptive resource provisioning for read intensive multi-tier applications in the cloud,” *Future Generation Computer Systems*, vol. 27, pp. 871–879, June 2011.

[26] L. Wu, *SLA-based Resource Provisioning for Management of Cloud-based Software-as-a-Service Applications.* PhD thesis, Dept. of Comput. and Inform. Syst., Univ. of Melbourne, Australia, 2014.

[27] D. Mensce and V. Almeida, *Capacity Planning for Web Performance: Metrics, Models and Methods.* Upper Sadale River, NJ: Prentice-Hall, 2012.

[28] *Cloud Vendor Benchmark 2015: Part 1.* Boston, MA: Cloud Spectator, 2015.

[29] M. Mao, J. Li, and M. Humphrey, “Cloud Auto-scaling with Deadline and Budget Constraints,” in *11th IEEE/ACM Intl. Conf. on Grid Computing (GRID)*, 2010.

[30] L. Logeswaran and H. Bandara, “Performance, Resource, and Cost Aware Resource Provisioning in the Cloud.”.

[31] B. Dougherty, J. White, and D. C. Schmidt, “Model-driven Auto-scaling of Green Cloud Computing Infrastructure,” *Future Gener. Comput. Syst.*, 2012.

[32] G. Garg, R. Buyya, , and W. Li, “Revenue Maximization Using Adaptive Resource Provisioning in Cloud Computing Environments,” in *ACM/IEEE 13th Intl. Conf. on Grid Computing (GRID)*, 2012.

[33] A. Dastjerdi, *QoS-aware and Semantic-based Service Coordination for Multi-Cloud Environments*. PhD thesis, Dept. of Comput. and Inform. Syst., Univ. of Melbourne, Australia, 2013.

[34] C. Hung, Y. Hu, and K. Li, "Auto-Scaling Model for Cloud Computing System," *International Journal of Hybrid Information Technology*, 2012.

[35] R. Chi, Z. Qian, and S. Lu, "A Game Theoretical method for Auto-Scaling of Multi-tiers Web Applications in Cloud," in *4th Asia-Pacific Symposium on Internetware*, 2012.

[36] H. Goudarzi and M. Pedram, "Maximizing Profit in Cloud Computing System via Resource Allocation," in *31st Intl. Conf. on Distributed Computing Systems Workshops (ICDCSW)*, 2011.

[37] R. Calheiros, R.Ranjan, A. Beloglazov, C. D. Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, journal=SoftwarePractice and Experience, year=2011,,"

[38] D. Kliazovich, P. Bouvry, and S. Khan, "GreenCloud: A packet-level simulator of energy-aware cloud computing data centers," in *Global Telecommun. Conf. (GLOBECOM 2010)*, 2011.

[39] D. Mosberger and T. Jin, "httperf - a tool for measuring web server performance," *SIGMETRICS Performance Evaluation Review*, 1998.

[40] A. Bahga and V. K. Madisetti, "Synthetic workload generation for cloud computing applications," *Journal of Software Engineering and Applications*, 2011.

[41] "Faban: open source performance workload creation and execution framework." Available at `http://faban.org/1.3/docs/index.html` (2015/07/20).

[42] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, and D. Patterson, "Rain: A workload generation toolkit for cloud computing applications," tech. rep., EECS Department, University of California, Berkeley, 2010.

[43] M. Arlitt and T. Jin, "1998 World Cup web site access logs," 1998.

[44] "ClarkNet HTTP Trace." Available at `http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html` (2015/07/20).

[45] J. L. Hellerstein, "Google cluster data," Jan. 2010.

[46] "RUBiS Implementation." Available at `http://rubis.ow2.org/index.html` (2015/07/20).

[47] "RUBBoS: Bulletin Board Benchmark." Available at `http://jmob.ow2.org/rubbos.html` (2015/07/20).

[48] D. Garcia and J. Garcia, "Tpc-w e-commerce benchmark evaluation," *IEEE Computer*, 2003.

[49] W. S. et al., "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," in *CCA*, 2008.

[50] R. J. Hyndman, *forecast: Forecasting functions for time series and linear models*, 2015. R package version 6.2.

[51] "Welcome to STAT 510 Applied time series analysis." Available at `https://onlinecourses.science.psu.edu/stat510/` (2015/07/20).

[52] "How to identify patterns in time series data: Time series analysis." Available at `https://www.statsoft.com/Textbook/Time-Series-Analysis` (2015/07/20).

[53] N. Wagner, Z. Michalewicz, S. Schellenberg, C. Chiriac, and A. Mohais, "Intelligent techniques for forecasting multiple time series in real-world systems," *Intl. Journal of Intelligent Computing and Cybernetics*, vol. 4, pp. 284–310, Aug. 2011.

[54] G. Zhang, "Time series forecasting using a hybrid ARIMA and neural network model," *Neurocomputing*, vol. 50, pp. 159–175, Jan. 2003.

[55] H. Zou and Y. Yang, "Combining time series models for forecasting," *Intl. Journal of Forecasting*, vol. 20, pp. 69–84, Jan. 2004.

[56] R. Adhikari and R. K. Agrawal, "Combining multiple time series models through a robust weighted mechanism," in *1st Intl. Conf. on Recent Advances in Information Technology (RAIT)*, Mar. 2012.