WORKLOAD, RESOURCE, AND PRICE AWARE PROACTIVE AUTO-SCALAR FOR DYNAMICALLY-PRICED VIRTUAL MACHINES

Daham Positha Pathiraja

179340D

M.Sc. in Computer Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

March 2019

WORKLOAD, RESOURCE, AND PRICE AWARE PROACTIVE AUTO-SCALAR FOR DYNAMICALLY-PRICED VIRTUAL MACHINES

Daham Positha Pathiraja

179340D

This report is submitted in partial fulfillment of the requirements for the Degree of
Master of Science in Computer Science specializing in Cloud Computing

M.Sc. in Computer Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

March 2019

Declaration

I, D.P. Pathiraja, hereby declare that this is my own work and this report does not incorporate without acknowledgement any material previously submitted for the degree or diploma in any other university or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles of books).

Signature: …………………….   Date: ……………………….

Name: D. P. Pathiraja

I certify that the above candidate has carried out research for the Masters thesis under my supervision.

Signature: ………………. …   Date: ………………………

Name of the supervisor: Dr. H. M. N. Dilum Bandara

Abstract

Proactive Cloud auto-scalers forecast future conditions and initiate scaling response in advance leading to better service quality and cost savings. Their effectiveness depends on the forecast accuracy and penalty due to miss prediction. However, such solutions assume fixed prices for virtualized Cloud resources to be provisioned. Hence, they are unable to benefit from dynamically-priced resources such as Amazon Spot Instances which are introduced by Cloud providers to deal with fluctuating workloads cost effectively. Moreover, users have the risk of losing resources when the dynamically-adjusted market price of resources exceeds the user-defined maximum bid price. Therefore, proactive auto-scalers should also forecast market price of dynamically-priced resources to minimize the cost further while retraining service quality. However, predicting the market price (to set the maximum bid price) is quite complicated given highly varying workload and resource demands. We present a proactive auto-scalar for dynamically-priced virtual machines by combing the workload and resource prediction capabilities of an existing auto-scalar named InteliScaler, and a novel technique for forecasting Spot price. We retrieve Spot price history from Amazon and use it to forecast the future prices using Recurrent Neural Networks. Next, we selected the maximum price for a given decision window as the bid value to make Spot request. To demonstrate the utility of the proposed solution, we tested the performance of the enhanced auto-scaler using a synthetic workload generated using the Rain toolkit and the RUBiS auction site prototype. Proposed auto-scaler with dynamically-priced virtual machines reduced the total cost by ~75% compared the same auto-scalar with fixed priced instances. Moreover, no noticeable change in service quality was observed.

Acknowledgement

I owe my deepest gratitude to my supervisor, Dr. Dilum Bandara of Department of Computer Science, Faculty of Engineering, University of Moratuwa, for his invaluable support in providing relevant knowledge, advice and supervision throughout the project. His continuous guidance, inspiring advices and constructive feedback provided this project a great value and an encouraging background throughout the project. This would not have been possible without his expertise and remarkable support.

I take this opportunity to convey our gratitude towards my batch mates, family members and friends who helped me to do my best towards achieving success during the entire project time period.

Finally, I would like to thank my colleagues at Sysco Labs (Pvt.) Ltd for covering my work and helping me to balance the workload. Without them, this project would not have been possible.

Last but not least, I am grateful for all the people who supported me throughout this research in various means.

Table of Contents

Declaration i

Abstract ii

Acknowledgement iii

Table of Contents iv

List of Figures vii

List of Tables ix

Abbreviations x

1. Introduction 1

1.1. Background 1

1.2. Motivation 3

1.3. Problem Statement 5

1.4. Objectives 5

1.5. Outline 6

2. Literature Review 7

2.1. Types of Virtualized Instances 7

2.1.1. Reserved Instances 7

2.1.2. On-demand Instances 7

2.1.3. Dynamically-priced Instances 8

2.2. Proactive Auto Scaling 11

2.3. Handling Spot Instances 14

2.3.1. Predict Biddings 14

2.3.2. Bidding Strategies 15

2.3.2.1. Always Bidding Maximum Price 15

2.3.2.2. AMAZING: Constrained Markov Decision Process (CMDP)-based Dual-Option Bid Strategy 16

2.3.3. Checkpointing 16

2.3.3.1. Hour-boundary Checkpointing 17

2.3.3.2. Rising Edge-driven Checkpointing 17

2.3.3.3. Checkpointing with Adaptive Decision 18

2.4. Spot Price Prediction Techniques 18

2.4.1. Autoregressive Integrated Moving Average (ARIMA) model 19
2.4.2. Long Short Term Memory (LSTM RNN) model 20
2.4.2.1. Neural Networks 20
2.4.2.2. Recurrent Neural Networks 21
2.4.2.3. Long Short Term Memory (LSTM) Networks 22
2.5. Summary 23
3. Methodology 25
3.1. High-Level Architecture 25
3.2. Workload Predictor 27
3.2.1. InteliScaler Prediction Algorithm 27
3.3. Price Predictor 28
3.4. Prediction Store 28
3.5. Bid Price Selector 29
3.6. Summary 29
4. Proposed Solution 30
4.1. Workload Prediction 30
4.1.1. Determination of Contribution Coefficients 32
4.1.2. Selection of Models 32
4.1.3. Prediction Algorithm 33
4.2. Price Prediction 34
4.2.1. Retrieving Spot Pricing History from AWS 34
4.2.2. Applying Famous Prediction Models 36
4.3. Proposed auto-scaling algorithm 39
4.4. Summary 39
5. Performance Analysis 41
5.1. Implementation 41
5.2. Optimal Bid Selection Procedure 43
5.3. Simulation-Based Evaluation 45
5.3.1. Simulation Setup 45
5.3.1.1. Proactive Auto Scaling 45
5.3.2. Scaling Out with Homogeneous Units 47

5.3.3. Scaling Out with Heterogeneous Units 48
5.4. Empirical Evaluation 51
5.4.1. Experimental Setup 51
5.4.2. Quality of Service Measurements 56
6. Summary 61
6.1. Conclusion 61
6.2. Research Limitations 63
6.3. Future Work 63
References 65

List of Figures

Figure 2.1 : Spot price fluctuations of us-east-1.linux.c1.medium Instance type. 9

Figure 2.2: How Amazon EC2 charges per-hour price for using a Spot Instance.. 4610

Figure 2.3: System Level Architecture of InteliScaler... 12

Figure 2.4: Hour-boundary checkpointing.. 17

Figure 2.5: Rising-edge driven checkpointing. 18

Figure 2.6: When user's Spot instance is out of bid 19

Figure 2.7: Chunk of a Recurrent Neural Network (RNN). 21

Figure 2.8: An Unrolled Recurrent Neural Network (RNN). 22

Figure 2.9: RNNs perform well when we need more contexts. 23

Figure 2.10: RNNs do not perform well when we need more contexts. 23

Figure 2.11: Repeating module in LSTM. 23

Figure 3.1: High-level architecture of proposed workload and resource aware auto scaler for Spot instances. 27

Figure 4.1: Spot instance pricing history. 35

Figure 4.2: Sample describe-spot-price-history API response 36

Figure 4.3: ARIMA prediction of Spot price for Spot instance type r5.xlarge.......... 37

Figure 4.4: LSTM RNN prediction of Spot price for Spot instance type r5............. 38

Figure 5.1: Simulation Setup... 41

Figure 5.2: Decide the best bid value for m4.4xlarge & m4.2xlarge Spot instances when scaling decision is made. 44

Figure 5.3: Penalty factor over violation percentage. 46

Figure 5.4: Resource Allocation 47

Figure 5.5: Variation of Virtual Machine Count and Load Average Overtime with Novel Approach for Scaling Out with Homogeneous Units for Confident Interval 6 hours.. 47

Figure 5.6: Variation of cost with time for different auto-scaling solutions when Spot confident interval is 6-hours... 48

Figure 5.7: Variation of Virtual Machine Count and Load Average Overtime with Novel Approach for Scaling Out with Heterogeneous Units for Confident Interval 6 - hours.. 49

Figure 5.8: Variation of cost with time for different auto-scaling solutions when Spot confident interval is 6-hours... 50

Figure 5.9: Simulation setup that mimic InteliScaler 53

Figure 5.10: Simulation setup that mimic novel auto scaler for Spot instances 54

Figure 5.11: Average response time of selected endpoints of RUBiS web application …. 60

List of Tables

Table 5.1: Violation Cost, Total Cost and Violation Percentage after 7-hours with Novel Approach for Scaling Out with Homogeneous Units for Confident Interval 6 -hours 47

Table 5.2: Violation Cost, Total Cost and Violation Percentage after 7-hours with Novel Approach for Scaling Out with Heterogeneous Units for Confident Interval 6 -hours 51

Table 5.3: Violation Cost, Total Cost and Violation Percentage after 7-hours for different auto-scaling methods 51

Table 5.4: Workload profile for simulation set-up 56

Table 5.5: Results in the simulation setup for mimic InteliScaler 56

Table 5.6: Scaling Decisions – Iteration 1 57

Table 5.7: Scaling Decisions – Iteration 2 57

Table 5.8: Scaling Decisions – Iteration 3 57

Table 5.9: Results of mimic novem price and workload aware auto sacaler for Spot Instances 57

Table 5.10: Scaling Decisions – Iteration 1 57

Table 5.11: Scaling Decisions – Iteration 2 58

Table 5.12: Scaling Decisions – Iteration 3 59

Abbreviations

APE	Absolute Percentage Error
ANN	Artificial Neural Network
ARIMA	Autoregressive Integrated Moving Average
ARMA	Auto Regressive
AWS	Amazon Web Services
CMDP	Constrained Markov Decision Process
EC2	Elastic Compute Cloud
IaaS	Infrastructure as a Service
IOPS	IO Operations per Second
LA	Load Average
MAPE	Mean Absolute Percentage Error
PaaS	Platform as a Service
POC	Proof of Concepts
PTPM	Price Transition Probability Matrix
QoS	Quality of Service
RMSE	Root Mean Square Error
RUBiS	Rice University Bidding System
VM	Virtual Machine

1. Introduction

1.1. Background

Cloud computing can be regarded as a model which is capable of providing network access to a shared pool of configurable computing resources in a ubiquitous, convenient and on-demand manner. Servers, applications, networks, storage and services are examples for such computing resources which can be provisioned quickly and released without any management effort or any intervention of the service provider.

Elasticity along with auto scaling enables virtualized cloud resources to be provisioned and de-provisioned as and when needed. The key goal of auto scaling is to provision resources as and when needed. It optimizes the resource utilization while minimizing the costs and providing higher levels of Quality of Service (QoS). To accomplish the optimization process, an auto-scalar should face two challenges; namely being aware of the workload that should be dealt with and provisioning the correct amount of resources in a cost efficient manner while preserving its desired levels of QoS.

An auto-scalar could follow two alternatives while dealing with the first challenge [1]. Either it may get a measure of upcoming workload using a reactive approach or it may use a proactive approach. In reactive approach an auto-scalar is interested in a selected set of events and thresholds. Events and thresholds can be specified at the point in which provisioning decisions should be taken. To follow a reactive auto-scaling system the user who maintains the cloud application needs to have expertise knowledge on thresholds. For instance, a user may define a rule such as spawning a new instance when memory reaches 80%. Sometimes these thresholds can be a combination of various other resource utilization limits such as CPU utilization, open file count, and pending queue size. In addition, mapping application matrices such as response time and throughput to CPU utilization and system-level metrics such as I/O Operation per Second (IOPS) is non-trivial. Therefore, expecting such level of knowledge about thresholds from user's side is not something suitable for an auto-scaling system [1].

Whereas a proactive approach never waits until such events to be triggered. Instead, it should have the ability to predict the workload ahead and take decisions based on those predictions. While reactive approach is easier to implement, when it comes to resource allocation, it does not perform well as it misses a considerable portion of the solution space. Even though a proactive approach seems very effective, the implementation of such a model is non-trivial. For example, a PaaS (Platform as a Service) system may come up with an auto-scalar which performs well on seasonal workload changes, but it may never be the well suited for taking decisions on frequently fluctuating loads. Therefore, having a precise understanding about future workload will also result in an efficient resource management. Best examples are spinning up VM instances in a proactive manner and avoid terminating them until end of billing cycle.

Spinning out virtual machines in a proactive manner and avoid terminating them at the middle of the billing cycle are good examples [1].

Through *on-demand instances* in cloud computing, the consumer is provided the capability to do the payments hourly or every minute, only for the consumed capacity without expecting any ahead of time charges [1]. According to the demand of the applications the consumer is capable of expanding or shrinking the compute capacities by just committing to hourly or once a minute payment. Whereas Spot *Instances* is a pricing model that enables users to bid on unused Amazon Elastic Compute Cloud capacity at whatever price user choose [4]. Such pricing models are introduced by the cloud providers to deal with fluctuating workloads in a cost-effective manner. According to historical values, the Spot price has been always higher when comparing with the price of on-demand instances. As a ratio it has been 50% to 90% from on-demand price [2]. Therefore, such pricing models could substantially reduce the cost of hosting applications on the cloud if utilized effectively. When a bid exceeds the Spot price, user gains access to the available Spot Instances and could run as long as the bid continues to exceed the Spot price.

Therefore, to continuously use a dynamic set of VMs while minimizing the overall cost, user needs to ensure bid price is dynamically adjusted and does not exceed the cost of on-demand price. However, predicting the maximum bid price is quite complex given varying workload and resource demands.

Techniques of making bids in an aggressive manner (i.e. make bids closer or slightly lower to the Spot market price) have been recommended by researches rather than using highest possible bidding values [3]. Optimal bidding strategies are available based on different theories and concepts. For example, Tang, Yuan, and Li [2] formulates the problem as a Constrained Markov Decision Process (CMDP) and proposed an optimal randomized bidding strategy through linear programming. They have done comparisons between various adaptive checkpointing mechanisms, by considering financial expenditure and time to complete tasks with the help of actual traces of price and workload models. They further evaluated their approach and came to certain decisions on how bidding should be handled properly on Spot instances to fulfill different goals with specific confidence levels. Moreover, out-of-bid scenarios should be well managed. Checkpointing is a mechanism which has been introduced to preserve the state of the applications while experiencing an out-of-bid failure scenario. Therefore, it is also important to consider both the bidding strategies and how to handle out-of-bid scenarios.

1.2. Motivation

Several related work focus on finding newer ways of auto-scaled cloud provisioning [1]. Reducing the expenses by getting rid of unnecessary provisioning and managing the already provisioned resources in the optimal way are the basic ideas behind these auto scaling approaches. There is a significant difference between pricing models of On-Demand Instances and Spot Instances [2]. Therefore, it is imperative to understand whether the same proactive auto-scaling mechanism can be applied for the dynamically-priced instances like Amazon EC2 Spot Instances as well.

It has been proven that proactive auto scaling approaches work more efficient and more cost effective than reactive auto scaling approaches in the context of PaaS systems. However, it does not necessarily mean that an IaaS system cannot be benefitted from proactive auto scaling. For example, InteliScaler [1] is a proactive auto scaler with the ability to provision cloud resources by analyzing workloads in a proactive manner. InteliScaler works under the assumption of static pricing model; hence, not designed for auto scaling dynamically-priced VMs such as Amazon Spot Instances. Moreover, it also assumes that nothing will cause any interruption towards the resources which are being consumed. This is not a big surprise because the pricing model and provisioning criteria are very different from on-demand instances. Therefore, it is imperative to extend the auto scaling ability of InteliScaler to support dynamically-priced instances. Such a solution will be equally important to IaaS systems as well. For example, it could enforce the efficient provisioning of dynamically-priced instances based infrastructure even with their intermittent nature of virtualized resources.

Several related work focuses on optimizing the bidding strategy and out of bid scenarios [2]. Most of those approaches enforce the user to make aggressive bids over higher bids and take actions to face failure situations with minimal damages. Checkpointing technique has been widely used as the state saving mechanism in most of the implementations of those approaches [4]. However, those methodologies are applied based on the assumption of static workloads.

Saving the state in an out-of-bid situation or knowing better bidding strategies are not sufficient to come up with a proper auto scaling mechanism for dynamically-priced instances. Without having a sufficient knowledge on the future variation of workload, it is hard to come up with a maximum utilization of resources for a given billing cycle. If it is possible to predict the Spot price fluctuation for a future time horizon, selecting a best possible value to be used as the maximum bid price in Spot request is possible.

1.3. Problem Statement

When considering above mentioned scenarios applying proactive auto scaling mechanism for dynamically-priced instances is not feasible unless we take their pricing models into consideration. To fulfill this, another sub-system is needed to manage bidding strategies and fail-over mechanisms. This sub-system must also have the ability to compare different type of dynamically-priced instances (of different sizes) to deal with dynamic workloads. Therefore, the research question to be addressed by this research can be formulated as:

How to apply workload, resource, and price aware proactive auto-scaling in the context of dynamically-priced instances?

1.4. Objectives

In order to provide the resolution to the above problem, following objectives should be achieved.

- To evaluate the effectiveness of existing proactive auto scaling techniques and bidding strategies in dealing with multiple dynamically-priced instance types.
- To extent suitable techniques or to develop new techniques to effectively predict bid prices of multiple dynamically-priced instance types.
- To extend the capability of InteliScaler to handle dynamically-priced instances for making predictions by proactively analyzing the workloads and prices.
- To evaluate the performance of the proposed solution using simulation and experimentations based on Amazon Spot Instance pricing.

1.5. Outline

Chapter 2 presents the literature review where it presents an overview of various Spot instance types, InteliScaler design, and dealing with Spot instances using techniques such as check pointing and sophisticated bidding strategies. Chapter 3 presents the methodology that we propose by explaining the high-level architecture of our approach. The major components of our novel methodology such as workload prediction and Spot price prediction are presented in Chapter 4. Chapter 5 presents how we have done the performance analysis by describing the simulation-based evaluation and empirical evaluation. Conclusion, Research Limitations and Future work are discussed in Chapter 6.

2. Literature Review

Auto scaling is a powerful concept that comes under cloud computing which provides the opportunity for users to scale up and scale down different cloud services such as virtual machines and service capacities accordingly with respect to each situation.

Cloud providers provision resources based on two major pricing models. Consumption-based pricing model and subscription-based pricing model are those two models. In Consumption-based pricing model customer pays according to the resources used and in subscription-based pricing model customer commits to the particular service for specified period of time.

Section 2.1 describes types of virtualized instances and how they are different from each other. The dynamically-priced instances are discussed in Section 2.1.3 explaining its pricing model and other important features. Section 2.2 presents auto-scaling solutions and an extensive discussion on InteliScaler. Finally, Section 2.3 describes the available approaches to deal with dynamically-priced instances.

2.1. Types of Virtualized Instances

2.1.1. Reserved Instances

As its name suggests this type of instances can be reserved and used with an additional confidence. The pricing model is straight forward where it is based on the resources and anticipated duration of use. Amazon recommends that this type of instances is well suitable for consumers will continuously use EC2 throughout one three years to make their computing expenditure lower and also for stable cloud applications [9].

2.1.2. On-demand Instances

This type of instances can be used by making the payments for compute capacity by the hour or second (minimum of 60 seconds). These instances do not include any upfront payments or long-term commitments. Consequently, user has the flexibility of changing the compute capacity according to the experiencing demand within a frequency of provider defined interval. In other words, the user can take compute capacity decisions in every provider defined interval. This pricing model is well suited for cloud consumers who are looking for much more flexibility from Amazon EC2 without any up-front payment or long-term commitment. For example, on-demand instances are well suited for fresh applications which use Amazon EC2 such as Proof of Concepts (POCs) [9]. Alternatively, the applications which are experiencing short-term unpredictable workloads can get tremendous advantages by moving into on-demand instances.

When we look at billing schemes of on-demand instances provided by different providers, significant changes can be identified. Amazon has a per-hour basis billing scheme. However, providers like Google Cloud and Azure publicize their capabilities of charging for partial hour [11]. Following is summary of popular billing strategies:

- AWS charges hourly basis starting from the first minute of the usage cycle.
- Azure charges for each minute with an amount of 1/60-th of hourly cost.
- Google Cloud charges 1/60-th of hourly cost and then charges for each minute after 10 minutes from the start.

Per-minute billing could be a good candidate if the user has a number of workloads whose durations are less than an hour [11]. For workloads which have durations more than one hour, per-minute billing is not effective and hourly price of the instance plays a significant role [11].

2.1.3. Dynamically-priced Instances

Dynamically-priced instances are widely used today to save cost [3]. User is capable of leasing extra resources when required. Various systems with computational tasks such as climate modeling, drug design, and protein analysis are good candidates for using dynamically-priced instances. Dynamically-priced instances can also be used in territories of data analysis like MapReduce. Rather than using a local HPC server, it is always beneficial to use dynamically-priced instances as it assembles cloud computing resources in on demand manner. Initially a user places resource leasing requests including the maximum amount (bid) that he/she is willing to pay per hour for any predefined instance type according to the preference. An instance will continue running since its prevailing market price is below the price which user has mentioned as the maximum price in the bidding request. However, the use of provisioning dynamically-priced instances is an efficient and cost-effective way which is really challenging. The reason is that the Spot price is not a function of either datacenter region, instance type, or operating system type of an instance. Moreover, in an occurrence of out-of-bid situation (i.e., Spot price goes beyond bid amount the spot) instance provider will terminate the running instance without any prior notice thus despite having economic benefits, this intermittent nature of dynamically-priced instances give no assurance about availability of VMs.

Figure 2.1 shows the price history of Amazon Spot Instance us-east-1.linux.c1.medium. It shows clearly that Spot instance price is highly subjected to periodic fluctuations based on supply and demand for their capacity. Amazon EC2 changes their Spot Price periodically due to two facts, namely when are new requests received and other is the changes of available Spot capacity due to instance terminations [2].

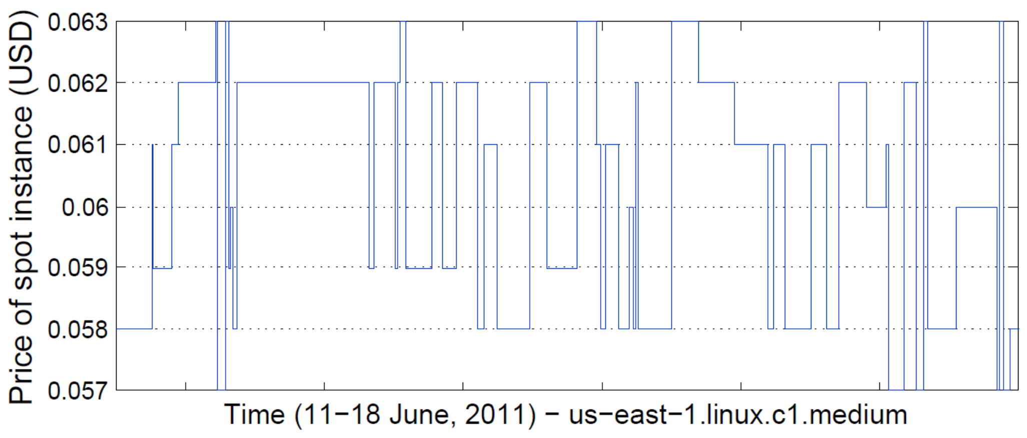


Figure 2.1: Spot price fluctuations of us-east-1.linux.c1.medium instance type.

Figure 2.2 depicts the way which Amazon EC2 bills hourly for the usage of Spot Instances [2]. As long as the user's highest bid is above the current Spot Price, Amazon grants the resources that are requested. However, Amazon terminates those resources immediately without notifying when current price exceeds the user's bid. The latter failure is called an out-of-bid situation. Here Amazon takes the responsibility and therefore, no payment is required for the partial hour. If user is the one who terminates the execution, he/she must settle the charge of the partial hour. To enhance the efficiency of task execution by checkpointing, consumers are advised to keep on persisting intermediate figures.

Even though the details mentioned above correspond to AWS Spot Instances we can see the same concept is being used by other cloud providers with different deviations. Google Cloud's "Pre-emptible VM Instances" are similar to Spot Instances in AWS. Here Pre-emptible VM Instances are 80% cheaper than a regular VM Instances but they are terminated after 24-hours. However unlike in AWS Spot Instances there is no variable pay in Pre-emptible VM Instances and the price is fixed. On the other hand, the Spot instance termination is not unmodified like in AWS. Consumer gets a 30-second time slot before terminating Pre-emptible VM Instances. To perform cleanup tasks, export logs and gracefully terminate a running process in the termination window, consumer is able to use Google shutdown script.

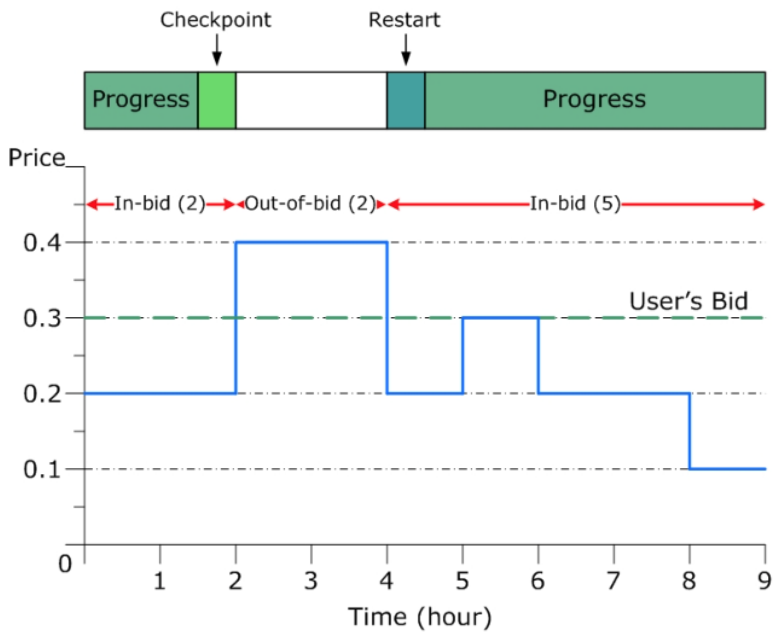


Figure 2.2: How Amazon EC2 charges per-hour price for using a Spot Instance.

Azure also introduced the concept of dynamically-priced instances with the name *Low-Priority VM* recently but still in public preview. The user is provided the capability of consuming the excess amount of compute power in the Azure data center pool via Low-Priority virtual machines. They can be only purchased via the Azure Batch Service. Low-Priority VMs have no concept of termination or notifying consumer about termination. Consumers can accommodate their batch jobs with low-priority virtual machines while moving into dedicated virtual machines whenever capacity goes down passing the lower bound of the threshold.

2.2. Proactive Auto Scaling

There are two types of auto scaling. Reactive auto scaling and proactive auto scaling are these two types. In reactive auto scaling type, scaling decisions are taken based on predefined thresholds for different workload metrics [3]. These predefined thresholds should be higher enough to cater efficient utilization of resources and also lower enough to compensate the additional time taken for the formulation and execution of scaling actions. One may think that this method is the best from two auto scaling methods because of its higher customizability from user's perspective. However, this reactive approach consists of several weaknesses such as inadaptability to different workload patterns, low resource consumption and high cost under smaller thresholds, and the risk of Quality of Service deterioration under larger thresholds and rapidly increasing workloads [4]. Therefore, without having an application-specific experience and expertise, deciding optimum thresholds for a reactive auto scaling system is not trivial.

Apache Stratos [1] is a good example towards automatically scaling infrastructure based on anticipated workloads rather than looking at predefined thresholds or events. This open source solution is a Platform as a Service which is capable of providing multi-tenancy and functionalities of multi-cloud deployment. Auto-scaling policy of Stratos can be regarded as proactive, as it takes scaling decisions based on forecasted workload for a future time horizon. Its forecasting algorithm considers the workload anticipated in the next minute. However, Stratos has several limitations such as small prediction timespan, limited reference to past data, reliance on user-defined threshold values, and focus only on homogeneous instances [1].

InteliScaler is a proactive auto-scaling solution which can be considered as a combination of three models; model of doing predictions, model of cost calculation, and a model for coordinating smart terminations. Predictive model forecasts workloads based on time-series forecasting and other machine-learning techniques. Cost model helps to find optimal solution for the resource allocation problem. InteliScaler uses an error-based ensemble technique to mix up multiple time-series analysis-based prediction algorithms to get a more accurate prediction of a workload. This prediction algorithm has the adaptability for the accumulation of dataset with time. However, as per the current implementation, InteliScaler supports only on-demand or reserved instances. Hence, it is imperative to explore how to extend InteliScaler to support dynamically-priced instances.

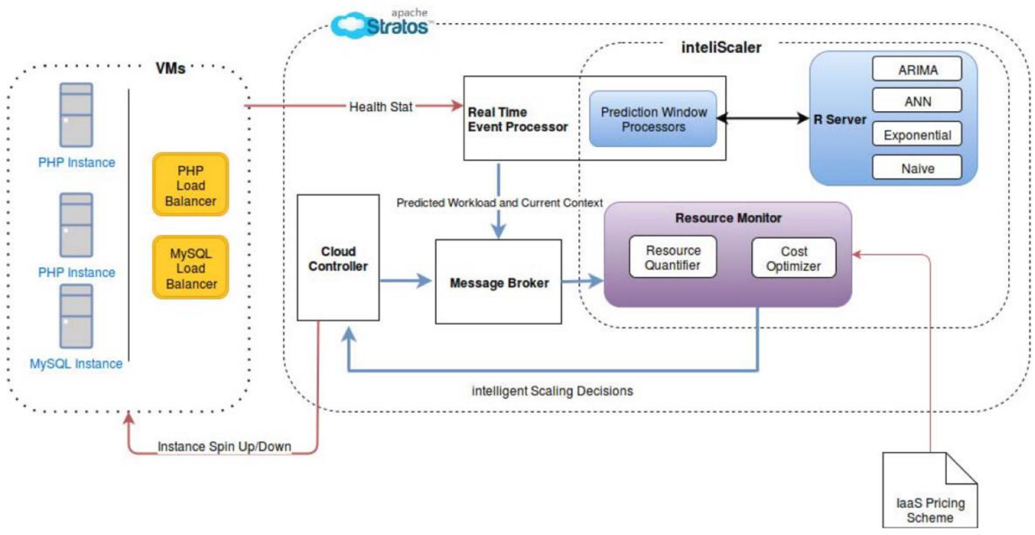


Figure 2.3: System Level Architecture of InteliScaler.

As illustrated in Figure 2.3, the high-level architecture of InteliScaler consists of two major components. One component is the Workload Predictor, which makes predictions about the future workload by using statistical data which is retrieved from the real-time event processor. These predictions are transferred to the other major component called Resource Monitor, which is responsible for provisioning (allocating and deallocating) resources according to the predictions received from workload predictor. Within resource monitor two sub-modules can be found. They are Resource Quantifier and Cost Optimizer. As its name suggests the resource quantifier is responsible for deciding the number of VM instances that are needed to control the forecasted workload whereas cost optimizer is the component which takes qualitative decisions. It can look at the pricing models of underlying Infrastructure as a Service (IaaS) and takes spin-up and spin-down decisions of VM instances [1].

InteliScaler does adaptive workload prediction through an error-based ensemble technique. This approach is capable of handling initial phase where no historical data is available and also the subsequent phase where data gets accumulated and becomes feasible to do offline training.

The prediction algorithm is capable of adapting the newer forecasts that it comes across, to the recent characteristics of workload data that has been accumulated. However, one significant flaw in almost all the prevailing error techniques is the inability of considering the global accuracy while emphasizing the error of the final prediction and assigning equal significance to the rest of the last prediction errors. InteliScaler addresses this particular problem by giving a high priority for the predictions which are more recent and a low priority for the earlier ones with a method called exponential smoothing [1].

The ensemble method of models has been enhanced such that all models collectively cover up wide range of characteristics of workloads. A model called Autoregressive Integrated Moving Average (ARIMA) is used to gain this capability [1].

The proposed prediction algorithm works as follows:

- Get the prediction value for the i-th model by taking the time series history at time *t* through ensemble setup.
- Using the i-th model, the history window for the recent *t* data points is set.
- Use an exponential smoothing model to fit the errors resulting from above step, and use them to calculate the contribution factor for the *i*-th model at *t*.
- Calculate the point forecast for time ($t + 1$) by summing up and averaging predicted values of each model by using above calculated contribution factors.

- Get the actual value for the time in (t + 1) and repeat the process to calculate value for ($t + 2$)

The scaling algorithm of InteliScaler takes both Quality of Service and price factors into consideration when quantifying resources. Instead of taking service up time as a QoS factor, InteliScaler goes into the deep by regarding any performance degradation (e.g., memory consumption, CPU utilization, and requests in flight) as a violation. Based on a formula that uses this idea it decides the most appropriate value that decreases the total cost to the lowest possible [1].

By the time of auto-scaling decisions are made by the InteliScaler, pricing scheme of the underneath Infrastructure as a Service layer is taken into consideration. It adapts the smart killing feature proposed by Bunch et al. [10] after evaluating the concept. The per-hour based billing system of on-demand instances lets InteliScaler to go ahead with that approach successfully.

2.3. Handling Spot Instances

2.3.1. Predict Biddings

Failures due to out-of-bid situation result inability of achieving required QoS of the cloud application. Therefore, most users follow the approach of bidding for highest possible value. However, there are significant advantages in using strategies of making aggressive bids which decide bids which are slightly below the current Spot market price.

The functionality of Amazon EC2 on-demand instances is equivalent to that of Spot instances despite their pricing model works in a different way. Setting higher bids for Spot instances with the intention of achieving higher availability is not a wise decision because in most of the cases it has been examined that Amazon uses a price closer to bid value for their on-demand instances. Therefore, there is no point in using a Spot instance for a price for which a more reliable on-demand instance can be purchased easily. As mentioned in [3] within a period of about 100 days from July

5 to October 15, 2011, Spot instance prices had gone beyond on-demand prices multiple times among most instance types and datacenters.

Another reason for users to be enforced using aggressive bidding strategy is the auction model of Spot Instances. For instance, Amazon auction model is very much similar to a Vickery auction (or Second price auction). Here users submit their bids in a sealed manner such that each has no idea about the bid price of the other. The Spot instance provider collects all the bids and calculates the winning-bidder who has the highest bid. The price which should be paid by the winner is the value of the lowest winning bid (sometimes, the highest non-winning bid). Therefore, users should always use fair prices for bidding to the benefit of all the users [4].

2.3.2. Bidding Strategies

The main intention of using bidding strategies is to minimize monetary cost based under a given reliability level. Tang, Yuan, and Li [2] described a simple but efficient heuristic together with their novel optimal bidding strategy based on Constrained Markov Decision Process. Following are a set of bidding strategies which minimizes the monetary cost under required reliability level. First one is Always Bidding Maximum strategy which is simple yet efficient heuristic. Next, another sophisticated strategy for making bids called AMAZING is proposed. AMAZING has the Constrained Markov Decision Process as its backbone.

2.3.2.1. Always Bidding Maximum Price

This is a heuristic bidding strategy. The implementation of this heuristic is quite easy and shows good performance when it is applied in practice. As the name suggests it always bid the maximum price without regarding the current state. This strategy assumes that if it always bid at the maximum price, Amazon will never terminate it. This heuristic has enhanced the work completion time by maintaining a constant execution. However, when looking at the current organization of Spot Instances, it is significant that highest price is at most 1.1 times larger than the lowest price. Therefore, Always Bidding Maximum Price (ABMP) will pay at most 10% more than the lowest cost.

2.3.2.2. AMAZING: Constrained Markov Decision Process (CMDP)-based Dual-Option Bid Strategy

AMAZING uses a constrained optimization problem based on a long-state evolving process. It defines several variables in its terminology. The variable *td* is the time taken for execution. When *td* and transition probability matrix are both provided, it starts finding the most appropriate bidding strategy μ in a manner which minimizes the expected monetary cost **E[M]** for the full job completion and expected execution time **E[Te]** that adheres to condition **E[Te]** $\leq$ *td*.

To provide a solution to the above optimization problem Tang, Yuan, and Li [2] have designed a novel strategy called AMAZING. This takes dynamic pricing model and the state transition intelligence into consideration. It has been constructed upon a back-end system called Price Transition Probability Matrix (PTPM). PTPM of a specific Spot instance type can be learnt from Spot Price history released on Amazon EC2 website.

As the resolution to above problem, authors have come up with an optimal bidding strategy which is capable of consuming both the dynamic pricing model and the state transition intelligence. The AMAZING strategy is constructed upon PTPM which can decide an appropriate bidding decision for the next Instance hour, one hour ahead of each, using a linear programming routine. We can construct an $\mathbf{E} \times \mathbf{E}$ matrix PTPM to represent the transition probability between each Spot Price pair σ and τ by using **E** different Spot Prices for a particular Instance type.

2.3.3. Checkpointing

Yi, Kondo, and Andrzejak [4] introduced a way of reducing monetary costs of computations in Amazon EC2's Spot instances with the help of check-pointing mechanism [5]. They have simulated and also compared the behaviors of different check-pointing mechanisms in terms of price and task completion time. Three of the check-pointing schemes mentioned in their work are discussed below.

2.3.3.1. Hour-boundary Checkpointing

As illustrated in Figure 2.4, in hour-boundary checkpointing method checkpointing happens periodically at each hour boundary. One reason for choosing 'hour' as the interval for checkpointing is that hour is the lowest granularity which can be achieved in Spot instance pricing. The other reason is that this scheme provides a guarantee of paying for the actual progress of computation. However, another variation of this approach can be found which does check-pointing periodically per every 10 or 30 minutes.

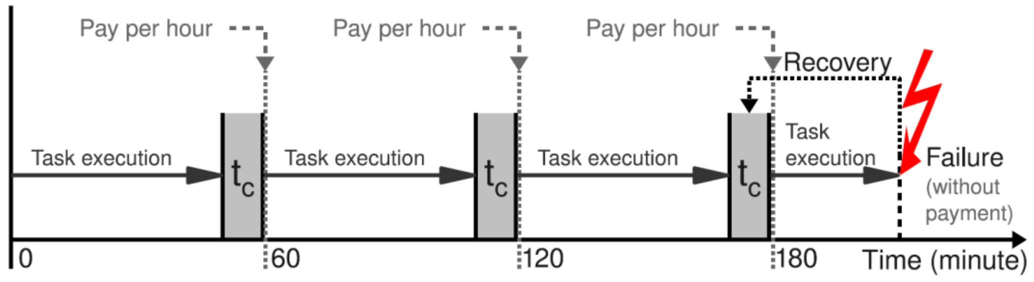


Figure 2.1: Hour-boundary checkpointing.

2.3.3.2. Rising Edge-driven Checkpointing

Rising edge driven check-pointing method (see Figure 2.5) is a newer kind of checkpointing method. In this method, the checkpoints are chosen in places where rising and falling edges are found according to the number of available resources, the bids from users, and the number of bidders. A rising edge is an indication of less available resources, more bidding users, and higher bids from users. Hence, it is more likely for an occurrence of out-of-bid event. However, it does not give the assurance of rising edge occurring at hour boundaries. Alternatively, it is more likely that no rising edge found with the period of availability. As a result, this method of check-pointing may result a failure at a situation of sudden increase of Spot price.

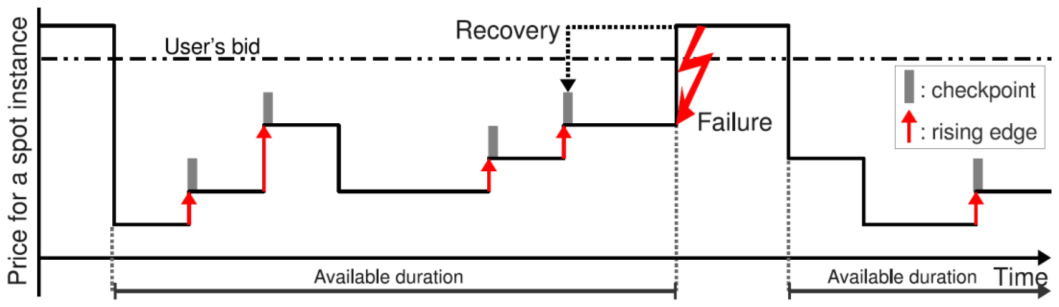


Figure 2.2: Rising-edge driven checkpointing.

2.3.3.3. Checkpointing with Adaptive Decision

Figure 2.6 illustrates how taking a checkpoint at the current time verses skipping a checkpoint at the current time affects the recovery time of the system when an out-of-bid situation or failure occurs. Figure 2.6 also shows how this selection effects affects the execution time of the current task. Yi, Kondo, and Andrzejak [4] also discussed by providing formulas whether it is useful to take a checkpoint at the current time or skip a checkpoint at the current time.

Yi, Kondo, and Andrzejak [4] have come up with 12 different check-pointing policies by combining all above-mentioned check pointing techniques which are orthogonal to each other. They have analyzed the impact of those policies on all 42 Spot instance types in Amazon EC2. Their results illustrate that the use of appropriate types of check-pointing reduces cost and task completion time.

2.4. Spot Price Prediction Techniques

It would be very useful for making estimations on maximum bid value in Spot requests if Spot price fluctuation can be predicted for a given future time horizon, in a similar way that InteliScaler predicts the future workload. We will first look at ARIMA model which is a time series analysis technique. The main intention of time series analysis is doing value predictions for a given time horizon in the future and figuring out recurrent patterns in the predicted series. We further researched on Neural Network based techniques because they are suitable for predicting non-linear sequences more effectively.

2.4.1. Autoregressive Integrated Moving Average (ARIMA) model

Autoregressive Integrated Moving Average (ARIMA) model is a combination of two models called auto regression (of order p) and moving average (of order q).

Auto regression Model AR(p) – Here the desired prediction is generated using a considerable set of past values adheres to a linear relationship among them. The meaning of auto regression in this context is that the variable is being regressed with respect to its own value. Thus, an autoregressive model of order p can be written as:

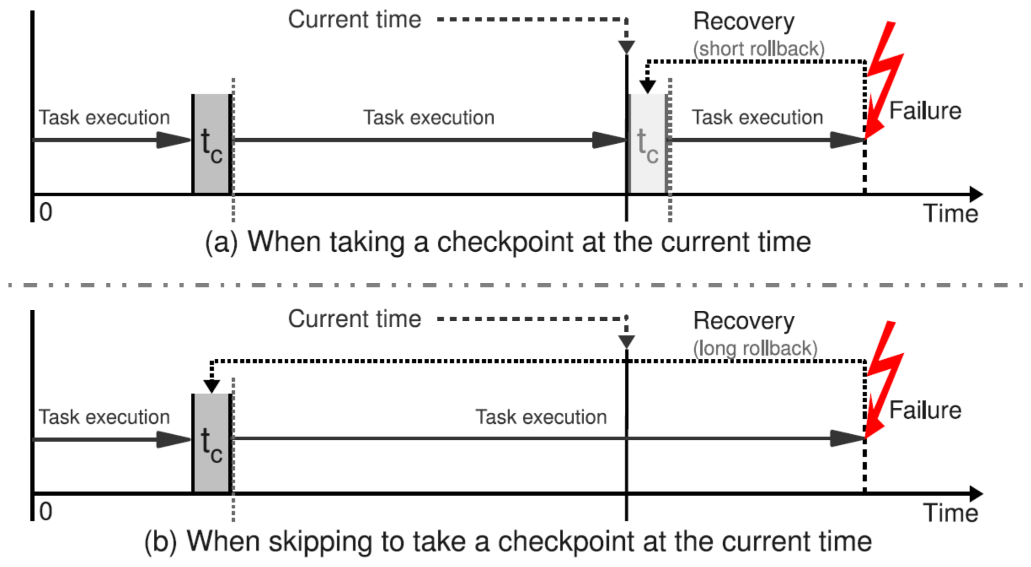


Figure 2.6: When user's Spot instance is out of bid.

$$x_t = c + \varphi_1 x_{t-1} + \varphi_2 x_{t-2} + \ldots + \varphi_p x_{t-p} + e_t \qquad (2.1)$$

where c is a constant and e_t is white noise. Auto regressive models are applicable for static data which has certain restrictions on the parameters [13].

Moving Average Model MA(q) – Here, previously predicted errors are being used in a model similar to regression model.

$$x_t = c + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \dots + \theta_q e_{t-q} \tag{2.2}$$

where e_t is white noise. x_t denotes the weighted moving average of most recent prediction errors. By combining differentiation of auto regression and a moving average model, non-seasonal ARIMA model can be obtained. The equation of the full model can be given as:

$$x'_t = c + \varphi_1 x'_{t-1} + \varphi_2 x'_{t-2} + \dots + \varphi_p x'_{t-p} + e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \dots + \theta_q e_{t-q} \tag{2.3}$$

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.

2.4.2. Long Short Term Memory (LSTM RNN) Model

2.4.2.1. Neural Networks

Artificial Neural Networks (ANNs) are a class of nonlinear and data-driven models which are inspired by the working mechanism of the human brain. This model can be regarded as the best alternative in the context of time series predicting where almost all statistical models require different assumptions to be satisfied to produce the best results. As even mentioned above neural networks are a good candidate for making predictions for time series which has nonlinear relationships. Neural networks vary depending on the topology. Hence, they are suitable for different machine learning tasks.

In the context of time series forecasting, a specific type of neural network category is being used. These are called autoregressive feed forward neural networks, which consists of a feed-forward structure and number of different layers. First, there is a layer for accepting inputs which is followed by a set of layers called as hidden layers.

Finally, there is another layer which emits output. Nodes of each layer is connected with the nodes of the layer adjacent to it via acyclic weighted edges. In an autoregressive neural network, lagged values of the time series are inserted as the inputs for making the model trained for a given time horizon with the help of historical dataset.

2.4.2.2. Recurrent Neural Networks

One shortcoming of traditional neural networks (ANNs) is the unclearness of the way that how it uses the reasoning about previous events to predict later. Recurrent neural networks address this issue. They are capable of storing information with the help of loops which they consist of. Figure 2.7 depicts a unit of neural network, A, which can output a value h_t by looking at some input x_t. Information is passed from one step of the neural network to the other via loops [14]. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Figure 2.8 shows what will happen if we unroll the loop in the RNN shown in Figure 2.7.

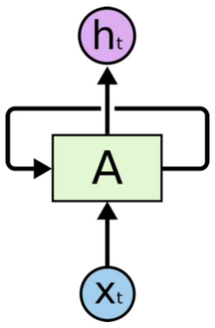


Figure 2.7: Chunk of a Recurrent Neural Network.

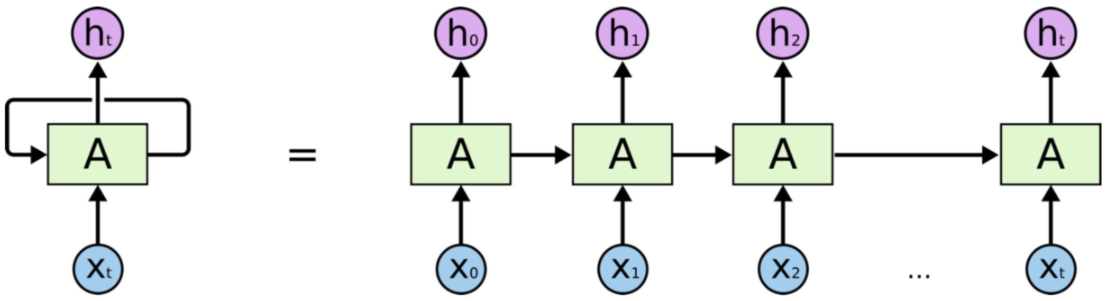


Figure 2.8: An Unrolled Recurrent Neural Network.

2.4.2.3. Long Short Term Memory (LSTM) Networks

RNNs can successfully be used to learn information in the past when the targeted information is only a short distance away from where it is being used. One example is a language model which tries to come up with the next word by taking earlier words into consideration. When the last word of "the clouds are in the sky," is being predicted, any further context is not necessary as figuring out the next word as "sky" is very obvious. Figure 2.9 depicts above behavior.

However, RNNs is not a good candidate (see Figure 2.10) when we need more contexts. In other words, LSTMs do not perform well when the targeted information is comparatively a large distance away from where it is being used. For instance, consider trying to predict the last word in the text "I grew up in France… I speak fluent French." Of course by looking at the recent information, figuring out that it is a name of a language is non-trivial. However, to figure out which language is that, we need the past context of France which is further back [14].

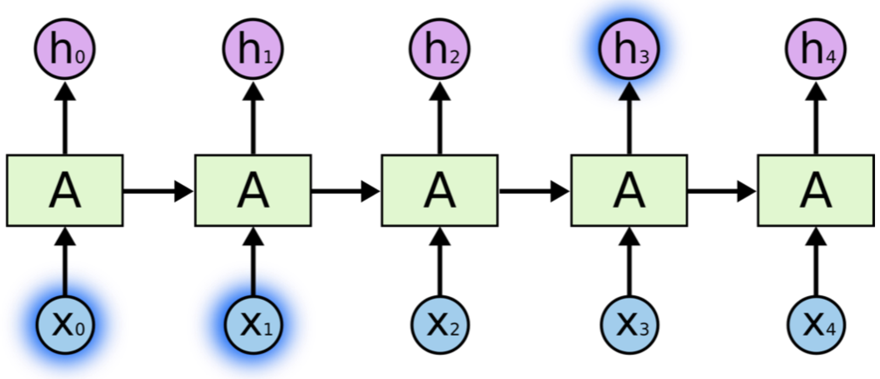


Figure 2.9: RNNs perform well when we need more contexts.

Long Short Term Memory (LSTM) networks) are a special implementation of RNN which is capable of learning about long-term dependencies. Hochreiter and Schmidhuber (1997) were the founders of LSTM but various people have done various amendments and improvements at later stages of the timeline. Their performance on a large variety of problems is immense and therefore widely being used. LSTMs are built to get rid of long-term dependency problem. They are anyway designed to remember information for long time durations. Hence, it is not something they struggle to learn. Figure 2.11 shows the repeating module in LSTM.

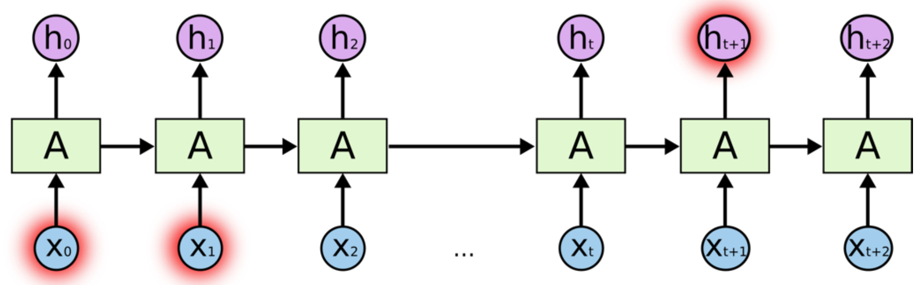


Figure 2.3: RNNs do not perform well when we need more contexts.

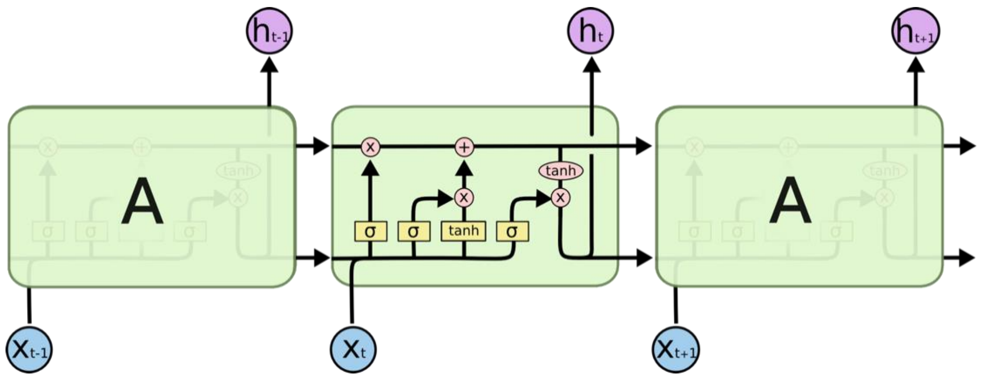


Figure 2.4: repeating module in LSTM.

2.5. Summary

AWS provides three types of VMs based on their pricing scheme, namely reserved instances, on-demand instances, and Spot instances. Reserved instances and on-demand instance types are fix priced VMs while Spot instance type is dynamically priced. Users should bid for Spot instances and the highest bid owner will get the instance if it is above the market price. Spot instances will remain until market price is below the bid price and will be terminated by the provider as soon as market price exceeds the bid value.

Proactive auto scaling takes scaling decisions based on an anticipated workload rather than scaling on predefined thresholds or events. InteliScaler is one such proactive auto-scaler which makes predictions for future time horizon using a special workload predictor which uses an ensemble of multiple prediction models. It also uses the concept smart killing for taking the maximum advantage of billing cycles of on-demand instances.

However, existing auto-scalers do not target dynamically-priced instances. Moreover, it is nontrivial to integrate it into an existing auto-scalar because of their intermittent behavior and difficulty in accurately forecasting Spot prices to make appropriate Spot requests. Several related work focus on bidding strategies to come up with better bidding requests. However, having a good bidding strategy is not sufficient to come up with a proper auto-scaler which should be aware on maximum resource utilization for a billing cycle or it does not give any idea on market price fluctuation of Spot instances over a given period in future time horizon.

Finally, we discussed about prediction techniques for making predictions about Spot price fluctuation in future time horizon. We looked at the same ARIMA model for time series analysis, as well as Neural Networks which work well with non-linear data sequences.

3. Methodology

Section 3.1 presents the high-level architecture of our novel auto-scaler for dynamically-priced virtual instances. The workload predictor used in our approach is presented in Section 3.2. Section 3.3 describes about Spot predictor while Section 3.4 and Section 3.5 describe prediction store and bid price selector, respectively.

3.1. High-Level Architecture

Today, various IaaS providers offer sophisticated pricing models which include beneficial resource usage schemes like the smart killing of instances and also there are auto-scalars which built upon them. However, it is hard to find a solution which gives that capability to PaaS level. Our benchmark model inteliScalar is such effort which has tried to address that territory. Low awareness on the workload can be regarded as a major limitation in prevailing PaaS setups. Therefore, good workload forecasting mechanism would enable PaaS auto-scalars to make more educated decision. InteliScaler consists of such workload prediction module which enables decision making. In this work we reuse InteliScaler’s workload predictor to make scaling decisions as it was well proven to be outperform existing techniques of other proactive auto-scalers. In other words, we do not need to replicate the existing functionality of making workload predictions using statistical data retrieved from the real-time event processor. Provisioning of Spot Instances should be done subsequently through an optimal bidding strategy. However, before deciding a bidding strategy it is important to identify which is our targeted Spot Instance type. This is because, unlike in On-demand Instances, of the frequent fluctuation in capacity and price tradeoff in Spot Instances.

To do the above subsequent steps when workload is being predicted we propose a setup comprised of model which can be referred when deciding the Spot instance type that we should bid upon. This model should be sensitive for market prices of various Spot instance types available. The optimal bidding strategy part can be reused or derived from various strategies which have been mentioned in the literature review.

We propose an automated scaling methodology, which is capable of forecasting the required bid value for making Spot requests based on anticipated workload, anticipated Spot price, and the minimum duration that Spot instances are available without termination. As shown in Figure 3.1, our solution has four major components each being individually responsible for one aspect of the scaling decision. *Workload Predictor* uses the same prediction mechanism used in InteliScaler. *Price Predictor* generates optimal Spot price predictions based on resource capacities and persists them. *Optimal Predictions Store* always contains the up-to-date optimal Spot price predictions, which are persisted by Spot Price Predictor. *Bid Price Selector* selects the optimal bid price according to the workload and optimal price predictions. Even though inteliScalar concentrated on PaaS models, our novel auto-scaler, especially the price and instance predictor, will be equally beneficial for IaaS models as well.

3.2. Workload Predictor

The error-based ensemble technique used in InteliScaler is reused as it provides good workload prediction. Its main focus is to consider scenarios where historical data is not present at the initial stage which results offline training infeasible. Ensemble method of workload predictor combines different prediction models come under machine leaning and time-serial analysis. With the accumulation of data the prediction algorithm is capable of adapting to newer predictions to the recent characteristics of which has already been accumulated. Overlooking the global accuracy and emphasizing the error of the most recent prediction is the common behavior of prevailing famous error quantification techniques, whereas the average errors of rest of all other last predictions get assigned equal significance. However, InteliScaler workload predictor requires assigning a larger significance to errors in more recent predictions and smaller significance to earlier predictions [1].

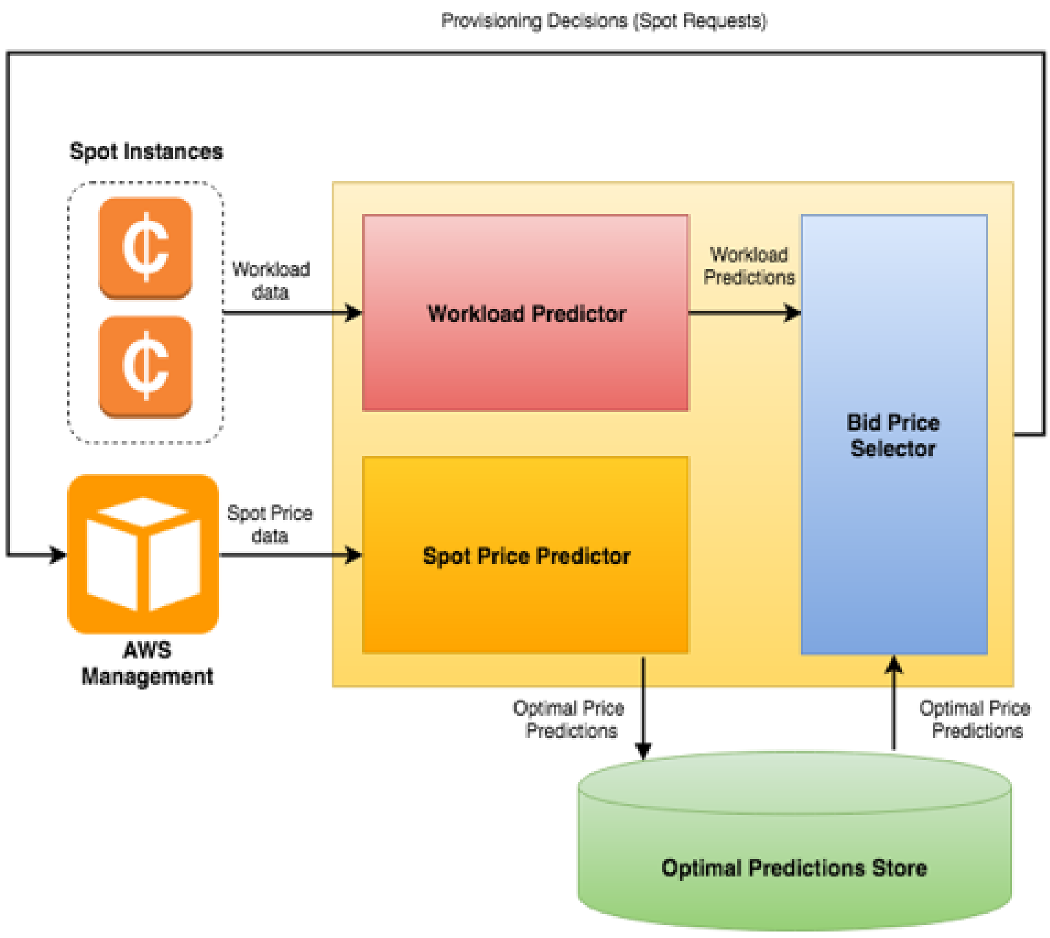


Figure 3.1: High-level architecture of proposed workload and resource aware auto scaler for Spot instances.

3.2.1. InteliScaler Prediction Algorithm

InteliScaler proposed workload prediction algorithm can be defined as follows:

1) Time series history window is taken at the time *t*, X = [x_1, x_2, ..., x_t]
2) Get the value predicted through the *i*-th time series forecasting method, throughout 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) Applying the *i*-th forecasting method position most recent *t* predictions in a history window.

4) With the use of exponential smoothing, the error values which are retrieved from above step will be fitted out. Those will be used to derive the contribution factor for the i-th model at t, $c_{(i,t)}$
5) Point forecast for time ($t + h$) will be derived from the following equation

$$\hat{x}_{t+h} = \frac{\sum_{i=1}^{k} c_{(i,t)} \hat{x}_{t+h}^{(i)}}{\sum_{i=1}^{k} c_{(i,t)}} \qquad (3.1)$$

6) Once real value is present for time *(t + 1)*, it is fed to the history window and proceeded from Step 2 again.

The algorithm has the flexibility of applying to any time-variant metrics such as request in flight, CPU or memory utilization and open file count because it does enforce any workload or performance metric comes under PaaS [1].

3.3. Price Predictor

The functionality of Price Predictor is to retrieve price history of a given dynamically-priced instance from the cloud provider and forecast Spot price by using that data as the training data set. For example, AWS exposes Spot Price through as a separate API. For predicting, initially we have used time-series based models because ARIMA like models performed well with most of the workload fluctuation patterns in InteliScaler. While the proposed architecture in Figure 3.1 can support any price prediction model, after experimenting various time series models, we identified that the LSTM RNN (Long Short Memory Recurrent Neural Networks) model to be more effective (comparison of different models is presented in Section 2.4). After taking the predictions for various instance types, it stores them in Prediction Store for the reference of Bid Price Selector.

3.4. Prediction Store

This is the storage which keeps various prediction from the Price Predictor. It is not necessary for updating prediction store in real time because we are not interested in the immediate price prediction but on the overall variation of Spot price throughout a future time horizon. Price Predictor may remain running as a parallel task and generate predictions to be stored. Because of the Prediction Store, Price Predictor has received the capability of offloading data sets to cloud and get them processed by giving the sufficient processing power.

3.5. Bid Price Selector

This module is triggered whenever the scaling decisions are made. Bid Price Selector has access to the all prediction series available in Prediction Store. To fulfill specific VM requirement it does not look at a single Spot instance type, but it looks at all possible instances combinations which are suitable to for catering the current vacancy.

3.6. Summary

The proposed auto-scaler for dynamically-priced VMs consists of four major modules. We reuse the same workload predictor of InteliScaler as its prediction model out performs existing prediction methods used in other proactive auto-scalers. We also came up with a price prediction model which is capable of providing predictions for Spot market price with a higher accuracy by taking Spot price history as the input. Prediction Store is used to persist predicted Spot prices for future reference by Bid Price Selector. By analyzing various predictions stored in Prediction Store, Bid Price Selector selects the most appropriate instance type at a given moment when scaling decisions are being taken.

4. Proposed Solution

In this section we discuss design details of the proposed proactive auto-scalar capability for supporting dynamically-priced VMs. Section 4.1 presents how workload prediction is performed in our proposed solution while prediction of Spot Instance market price for a time horizon in future is presented in Section 4.2.

4.1. Workload Prediction

We use the workload prediction model of InteliScaler as it is. Ensemble prediction model used in InteliScaler outperformed various famous forecasting techniques which are currently available. It has been compared with demographic techniques like ARIMA model, neural networks and exponential model [1]. For that comparison, various workload traces which are publicly available have been used. The workload predictor proposed in InteliScaler attempts to address the following challenges specific to workload prediction for auto-scaling:

- A Platform as a Service cloud system should be able to cater heterogeneous applications with heterogeneous patterns of workload. Therefore, overfitting on a specific workload pattern is something which should not happen from a proper workload predictor.
- With the continuous accumulation of workload data with the time, the predictive model should be capable of adapting to recent workload characteristics.
- The results should be produced by the workload predictor within a given time
- The prediction time horizon should be selected looking at several physical constraints such as uptime and graceful shutdown time of virtual machines. Therefore, it is essential for the workload predictor to come up with sufficiently accurate results throughout a sufficiently large time horizon.

Therefore, InteliScaler's prediction model is capable of training data in near real time to identify resent trends and drastically different workload patterns. As it was determined that individual prediction models cannot capture varying workload patterns, authors proposed an ensemble technique that combined the predictions from ARIMA, exponential smoothing, and neural networks. Then the final prediction was derived based on the error produced by each of the predictors. This approach focuses on scenarios where considerable amount of workload history data is absent at the early stages of the forecasting process and hence, offline training is impossible. With the time, auto scaler accumulates the workload history (e.g., CPU, memory, and request count). The model is capable of collecting most recent data in real time after the initial prediction and make it useful for the subsequent predictions by adding them into the workload history. Authors then used mean values of error metrics (e.g., absolute error, absolute percentage error, and squared error) to determine the contributing factors of individual prediction models.

InteliScaler consider the workload to be represented as a time series. Let **X** = [x_1, x_2, …, x_t] be the currently available workload time series up to time *t*. Then the predictors objective is to calculate the predicted workload at *t* + *h* (*h* > 0). Let the predicted value be $\hat{x}_{t+h}$, and the predicted value from the *i*-th prediction model be $\hat{x}_{t+h}^{(i)}$. We define $\hat{x}_{t+h}$ as a weighted sum of predictions from a set of model *k* as follows:

$$\hat{x}_{t+h} = \sum_{i=1}^{k} w\hat{x}_{t+h}^{(i)} \qquad \forall k \in \{1, 2, 3, ..., n\} \tag{4.1}$$

where w_i is the weight of the *i*-th forecasting model. There is an assumption which tells that the summation of weights is equal to unity to confirm that it is not biased [16]. Hence, a variable called c_i (referred to as *contribution coefficient*) is defined as the contribution from the *i*-th model. So the above equation can be rewritten as:

$$\hat{x}_{t+h} = \frac{\sum_{i=1}^{k} c_{(i,t)}\hat{x}_{t+h}^{(i)}}{\sum_{i=1}^{k} c_{(i,t)}} \tag{4.2}$$

where

$$w_i = \frac{c_i}{\sum_{j=1}^{k} c_j}. \tag{4.3}$$

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

4.1.1. Determination of Contribution Coefficients

Contribution coefficients of the ensemble technique are determined based on the past errors of each considered model. Also, the accuracy of next time horizon's prediction is highly significant in this context. Therefore, inverse values of past forecasting error measures are being considered when calculating the contributions.

Error measurement has various alternatives, and authors have considered mean values of error metrics such as absolute error, absolute percentage error, and squared error. The limitation of this approach is that, even though such techniques capture the level of accuracy in the last calculated prediction, it does not take the overall accuracy into consideration. This approach could go wrong in a scenario where the model has continuously produced huge errors with previous predictions except the latest prediction because the contribution can be reduced significantly even though it produces the best predictions for latest values.

4.1.2. Selection of Models

The error-based weighting mechanism in InteliScaler is very effective in dealing with drastically different workload patterns, as it captures different characteristics of those datasets using conceptually different models. For example, ARIMA model looks at the time series values with the assumption of values follow a linear correlation structure. Therefore, it is incapable of capturing non-linear patterns. It is possible to used seasonal ARIMA models to fit the seasonal factors of the time series with a higher accuracy.

Compared to the ARIMA model, a neural network is good at capturing nonlinear relationships of time series. Moreover, a neural network has the ability to extract patterns even without any prior knowledge of relationships in time series data as it follows a data-driven approach. However, to accurately identify patterns neural networks require sufficiently large data set. Therefore, even though a neural network makes larger errors in initial phases it performs well as the data gets accumulated with time. One other limitation of neural networks is the time taken for training. As we are interested in near real-time training, we cannot let the data point history to grow arbitrarily large.

Even though ARIMA is believed to be more generic than exponential models, in some specific occasions it is hard to find ARIMA model which behaves similar to the exponential models (e.g. the exponential trend model) [16]. Therefore, to preserve generality, exponential model is also integrated in to the ensemble model.

Producing an out of range prediction is a possible in any of the above models. For instance, it is highly likely that neural networks produce out of range predictions in early phases. In such occasions, we can compensate it by using the naïve forecast model which chooses data point which is known last as the next interval's prediction.

4.1.3. Prediction Algorithm

Given the prediction models and error metric, the workload prediction algorithm proposed in InteliScaler can be summarized as follows:

1. Time series history window is taken at the time t, X = [x_1, x_2, ..., x_t].
2. The value predicted through the i-th time series forecasting method, throughout the time horizon h, $\hat{x}_{t+h}^{(i)}$ is taken.
3. A history window is fitted for the last t actual data points using the i-th method.
4. To fit the errors resulting from Step 3 exponential smoothing model is used. The results were used for calculating the contribution factor for i-th model at t, $c_{(i,\,t)}$.
5. Following equation is used for the calculation of the point forecast for time $t + h$.

$$\hat{x}_{t+h} = \frac{\sum_{i=1}^{k} c_{(i,t)}\hat{x}_{t+h}^{(i)}}{\sum_{i=1}^{k} c_{(i,t)}} \tag{4.4}$$

6. The actual value for time t + 1 will be available at time t + 1. Add this value to the history window X = X ∪ {x_{t+1}}and repeat from Step 2.

4.2. Price Prediction

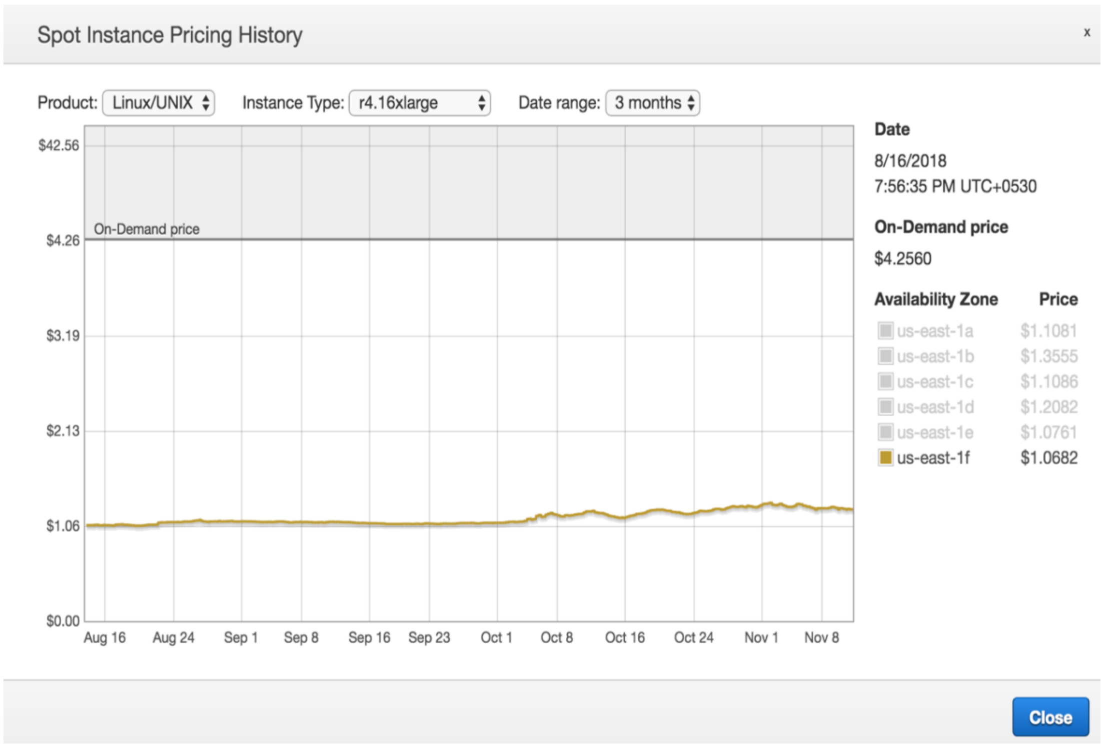
Price Prediction consists of three major steps. In the first step, price history for a given type of instance is taken from the cloud provider. For example, for AWS, EC2 Client provided by AWS SDK (Software Development Kit) could be used obtain price history within any given period up to three months. Typically, such SDKs support multiple programming languages; hence, could be used to most types of cloud-based applications. If data points are not given a time series (e.g., AWS gives only the Spot price fluctuations), they need to be converted to a time series. The next step is to forecast the instance price using a suitable prediction model. Initially, we used an ARIMA model, as it has been used even in InteliScaler for workload predictions. Afterwards, to overcome some major drawbacks in ARIMA model we used the Long Short-Term Memory (LSTM) model. LSTM is a popular Recurrent Neural Network (RNN) model which is essentially a nonlinear time series model, where the nonlinearity is learned from the data. Moreover, when compared to alternative RNNs, hidden Markov models and other sequence learning methods it shows tremendous performance due to its relative insensitivity to gap length.

4.2.1. Retrieving Spot Pricing History from AWS

As AWS Spot Instances are the only mature dynamically-priced VMs available as of today, we build out Proof of Concept (PoC) solution around it. AWS provides a SDK to obtain spot pricing history of a given EC2 Spot instance for the last 90 days. For example, Figure 4.1 shows how the price of a Linux r4.16xlarge instance in us-east-1f Availability Zones changes over 30 days. AWS recommends the user to review the Spot price history before specifying a maximum price when making a Spot request.

Amazon has exposed set of APIs for users to get various information that they are interested related to AWS instances. Some of them are capable of controlling the AWS infrastructure while some of them merely provide data on a particular subject. The API “describe-spot-price” is capable of describing the Spot price history.

When time range is selected, this API is capable of showcasing the price fluctuation of each type of instance. It should be kept in mind that the response contains only timestamps where the price has altered [12]. For instance, if user creates a proper request to “describe-spot-price” API user will get a response like in Figure 4.2. This response is an array of objects corresponding to the times where Spot price has fluctuated.


Spot Instance Pricing History

Product: Linux/UNIX
Instance Type: r4.16xlarge
Date range: 3 months

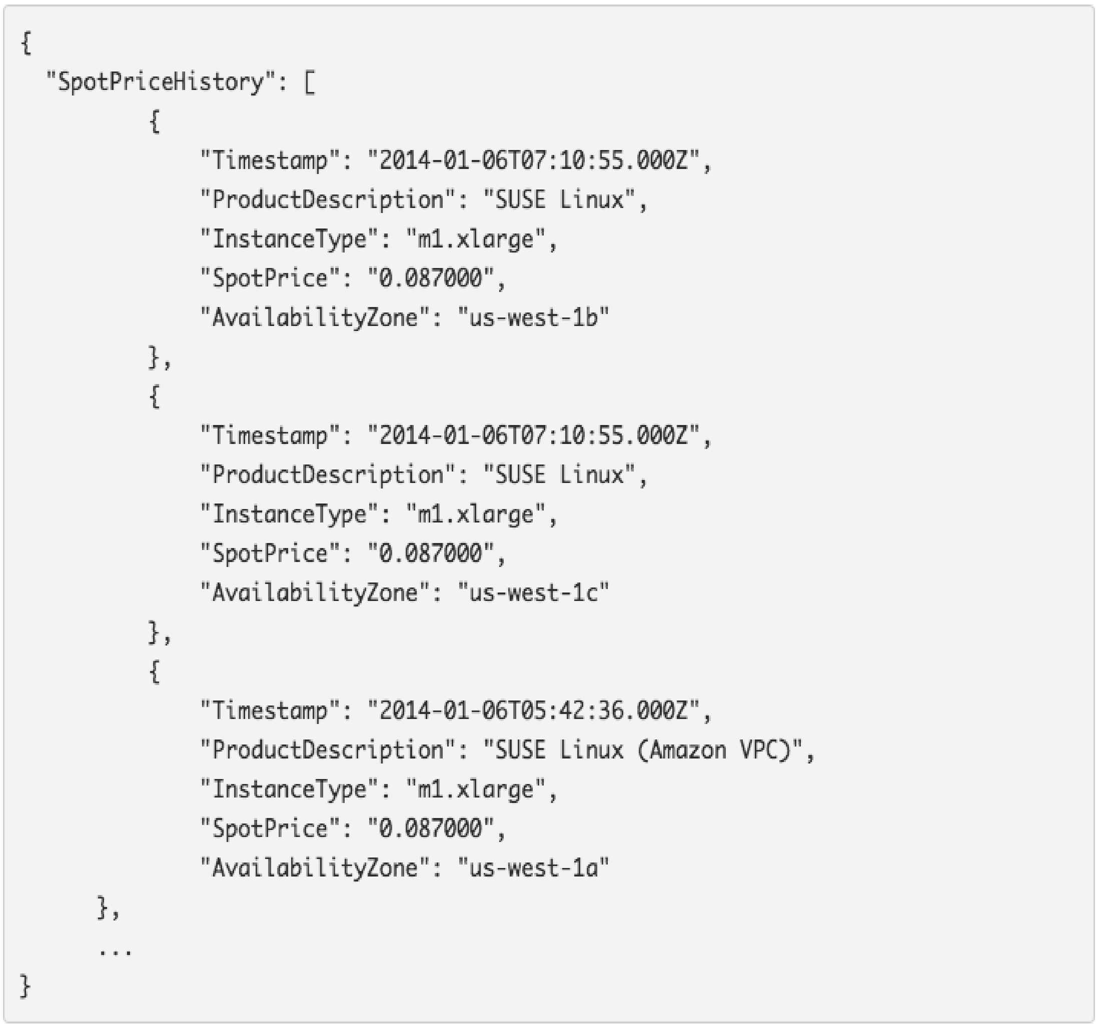
Date
8/16/2018
7:56:35 PM UTC+0530

On-Demand price
$4.2560

Availability Zone	Price
us-east-1a	$1.1081
us-east-1b	$1.3555
us-east-1c	$1.1086
us-east-1d	$1.2082
us-east-1e	$1.0761
us-east-1f	$1.0682

Figure 4.1: Spot instance pricing history.

The AWS SDK for Node.js consists of APIs which can be used to consume AWS services when implementing Node.js or browser applications. This is very sophisticated library which has covered the majority of AWS endpoints. In our implementation aws-ec2-buddy we have used AWS SDK which is available through NPM (Node Package Manager).



```
{
  "SpotPriceHistory": [
          {
              "Timestamp": "2014-01-06T07:10:55.000Z",
              "ProductDescription": "SUSE Linux",
              "InstanceType": "m1.xlarge",
              "SpotPrice": "0.087000",
              "AvailabilityZone": "us-west-1b"
          },
          {
              "Timestamp": "2014-01-06T07:10:55.000Z",
              "ProductDescription": "SUSE Linux",
              "InstanceType": "m1.xlarge",
              "SpotPrice": "0.087000",
              "AvailabilityZone": "us-west-1c"
          },
          {
              "Timestamp": "2014-01-06T05:42:36.000Z",
              "ProductDescription": "SUSE Linux (Amazon VPC)",
              "InstanceType": "m1.xlarge",
              "SpotPrice": "0.087000",
              "AvailabilityZone": "us-west-1a"
      },
      ...
}
```

Figure 4.2: Sample describe-spot-price-history API response.

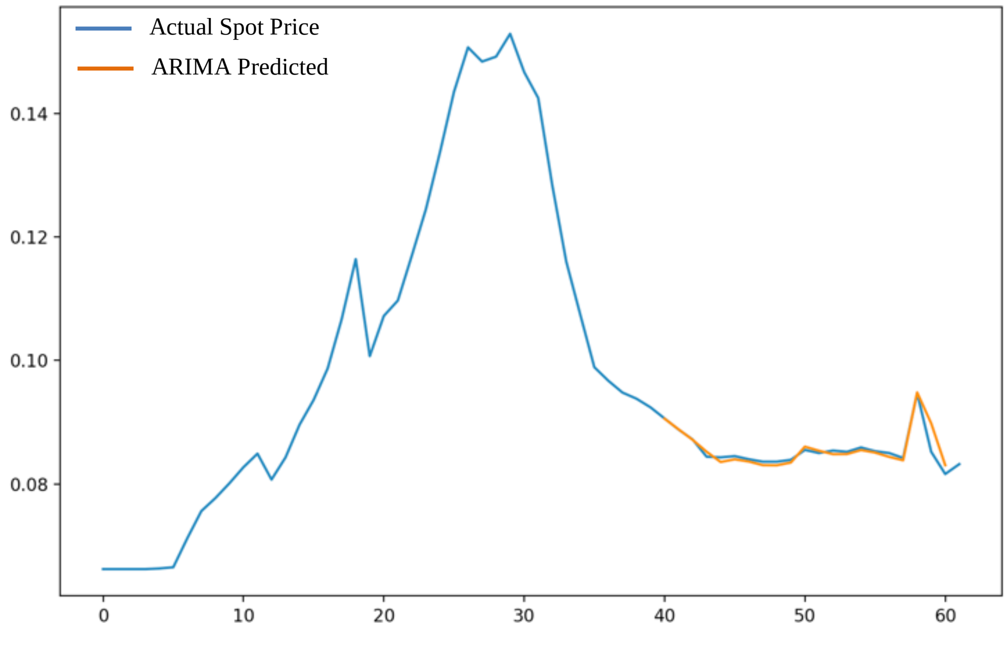
4.2.2. Applying Famous Prediction Models

Same time series forecasting techniques can be used to predict Spot price as well. However, it is hard to use the same series that we get as the response of describe-spot-history API because time series prediction algorithms need continuous data to be fed. The describe-spot-history API outputs Spot prices only for the points where price get fluctuated. Nevertheless, we were capable of generating a continuous Spot price history by applying padding values to the data set that we already have.

To predict Spot price, we have initially used Autoregressive Integrated Moving Average (ARIMA) model. Then we moved into another model called Long Short

Term Memory Recurrent Neural Networks (LSTM RNN) as ARIMA did not provide predictions with a very high accuracy.

We used ARIMA model in the initial stage as it has been used even in InteliScaler for workload predictions. However, when ARIMA has been used as the third step of our model for making predictions based on a given Spot history, being a non-linear complex structured model, it has not performed well. When we used one portion of some Spot history as the training data set and the rest as test data, we received considerably large Mean Absolute Error (MAE) values and Root Mean Squared Error (RMSE) values for ARIMA model. ARIMA model assumes linear relationships between independent and dependent variables resulting above observations thus ARIMA seems not suitable for forecasting due to its inability to model irregularities [5]. Figure.4.3 illustrates how ARIMA has predicted Spot price for Spot instance type *r5.xlarge*. Here 66% of the Spot history values have been taken as the train set whereas 33% of Spot history data has been used as the test data set to come up with error measures.



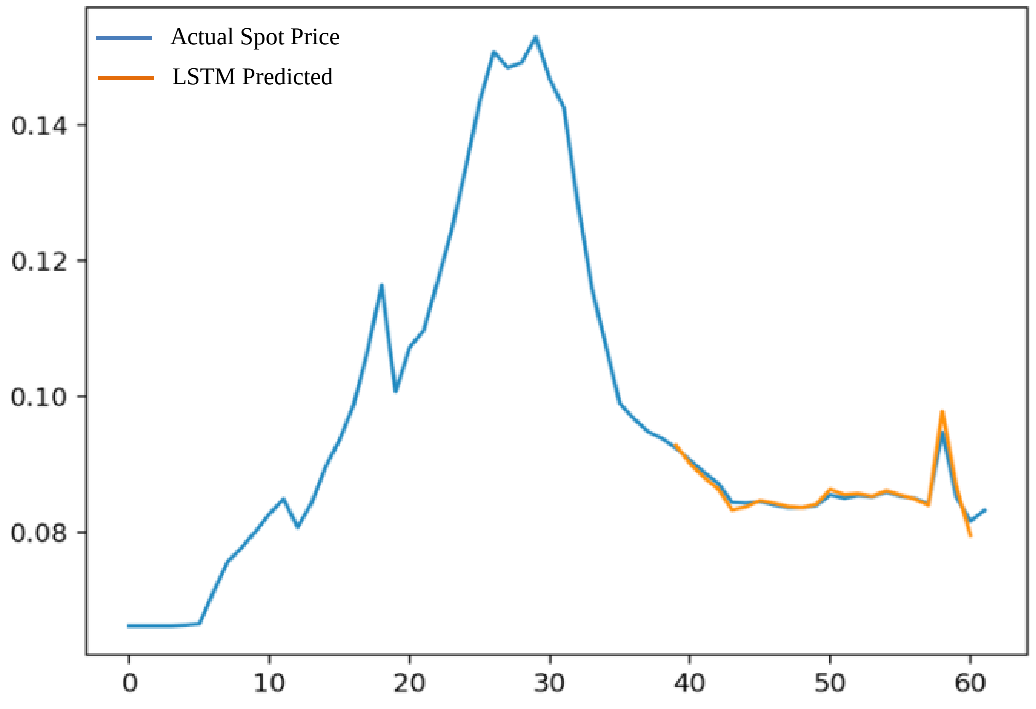
MSE	RMSE	MAE
0.000015	0.003900	0.046755

Figure 4.3: ARIMA prediction of Spot price for Spot instance type r5.xlarge.

Afterwards, to overcome some major drawbacks of ARIMA model, the Long Short-Term Memory (LSTM) model, which is a famous Recurrent Neural Network (RNN) model has been introduced to get better predictions than with ARIMA.

Because ARIMA model does not give predictions with a high accuracy using non-linear Spot price historical data, we tried out RNN model which is already known for its best performance in many load prediction problems. Compared to ARIMA, RNN model is powerful in giving predictions for a time series by taking long temporal dependencies into consideration. The magnitudes of these dependencies may be varying with time or unknown prior to the prediction time.

Figure 4.4 shows how LSTM RNN has predicted Spot price for Spot instance type r5.xlarge. Here 66% of the Spot history values have been taken as the train set and 33% of Spot history data has been used as the test data set to come up with the error measures.



MSE	RMSE	MAE
0.000001	0.001142	0.025674

Figure 4.4: LSTM RNN prediction of Spot price for Spot instance type r5.

When comparing the Root Mean Square Error (RMS) of LSTM RNN and ARIMA model based predictions (0.001142 and 0.003900) it is clear that the LSTM RNN model gives predictions which are more concentrated around the line of best fit. We also measured the Mean Absolute Error (MAE) for both LSTM and ARIMA predictions, to check the average over the test sample of the absolute differences between predicted values and real observations where each difference has same weight. Even this measurement of LSTM RNN model (which is 0.025674) out performs ARIMA (which is 0.046755) in a significant manner.

4.3. Proposed auto-scaling algorithm

The proposed Spot instance auto-scaling algorithm can be defined as follows:

1. Predict workload using InteliScaler workload predictor.
2. Predict Spot price for instances belong to same family using LSTM RNN based method.
3. When scaling up decision to be taken, calculate size of the Spot instance cluster needed to cater anticipated workload of the given time and come up with all permutations of Spot instances which can cater that particular requirement.
4. Decide which permutation is the most cost effective by getting their maximum possible price values within the given decision window. Use that price as the bid value for the Spot requests within that window.
5. When scaling down decision to be taken, remove required number of instances whose current Spot price is highest.

4.4. Summary

InteliScaler uses ensemble workload prediction model which was proven to outperform individual predictors based on statistical methods like ARIMA and exponential model, machine-learning models like neural networks, and the prediction method used in Apache Stratos. Therefore, we decided to use the same prediction model of InteliScaler as our workload prediction scheme. We carried out a number of experiments regarding price prediction to determine a model which provides prediction with a higher accuracy. Initially we tried with models such as ARIMA to predict the Spot price for a future time horizon using a large Spot history as a training set. As this did not lead to an accurate prediction, we attempted the Long Short-Term Memory (LSTM) model which has been introduced to get better predictions than with ARIMA. According to the statistical results we received after experimenting its capability of price prediction, LSTM clearly outperforms ARIMA model.

5. Performance Analysis

Section 5.1 describes the simulation setup for performance analysis. The optimal bid selection procedure of our proposed approach is presented in Section 5.2. Section 5.3 presents the auto scaling algorithm. Section 5.4 explains the simulation-based evaluation while Section 5.5 explains the empirical evaluation.

5.1. Implementation

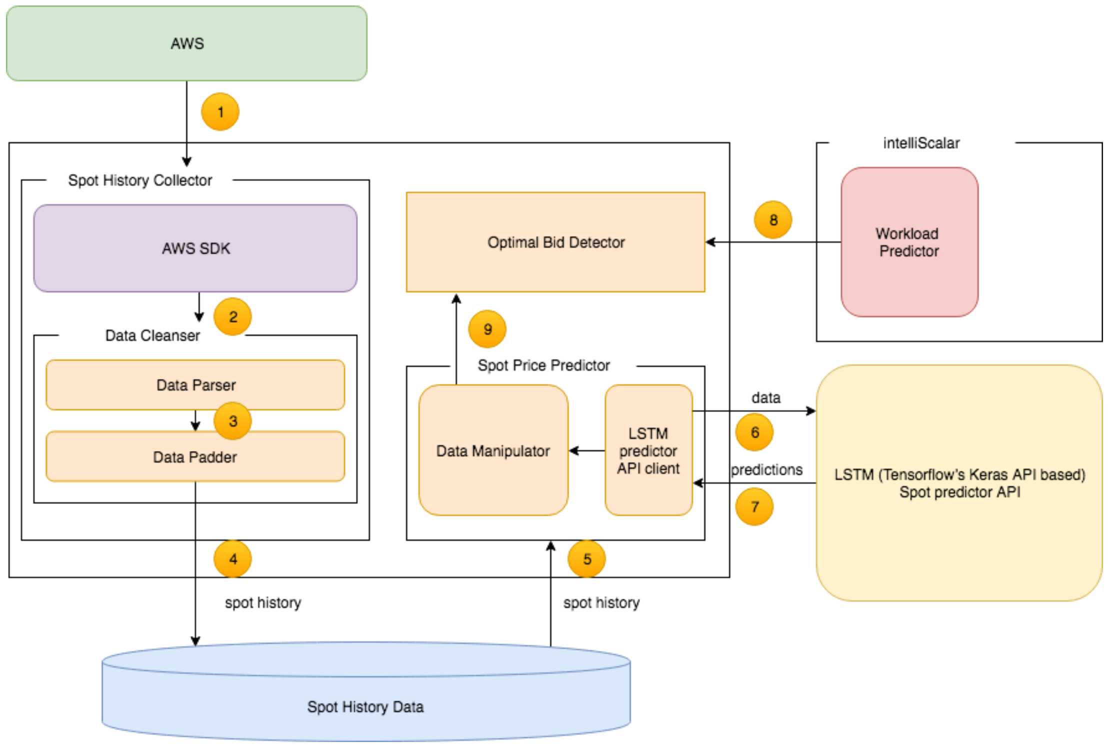


Figure 5.1: Simulation Setup.

As shown in Figure 5.1 our simulation model consists of four major models. Spot History Collector, Spot predictor API, Price Predictor, and Optimal Price Predictor are those four modules. Spot History Collector is a module implemented in Node.js to retrieve Spot history for a given instance type from AWS and manipulate that data in a manner which is eligible to be fed as a training data set for any time series prediction model. Price Predictor API is a Python implementation of LSTM model with the help of Tensor Flow's Keras API [16]. This API is hosted separately in an EC2 instance because it requires more resources to execute. Price Predictor is also another Python implementation which retrieves Spot history data from the storage which was stored by Spot History Collector and get predictions for given future time horizons. Optimal Bid Predictor is capable of detecting the optimal bid required to make a Spot request when scaling decisions are taken with the help of InteliScaler workload predictor and our novel LSTM RNN based Price Predictor.

As depicted in Figure 5.1, the simulation flow is as follows:

1. Retrieving Spot price history from AWS – To retrieve Spot price history using the describe-spot-history API, we use AWS SDK for Node.js. As we need the longest possible history of Spot price fluctuations, we fetch history of a particular instance type throughout a span of 3-months from that API which the maximum it offers.
2. Presenting Spot history to the Parser – The Spot price history obtained from describe-spot-history API need to be converted to a time series such that it can be fed into the prediction models. Thus, this parser transforms describe-spot-history API response into a time series.
3. Presenting parsed Spot history to Data Padder – As describe-spot-history API does not provide a Spot history with uniform time gaps (i.e., hours), we further need to make sure the derived time series is equally spaced in time before feeding it into the time series prediction models. Because AWS provides only the points which Spot price got fluctuated, we can easily derive the price of intermediate time points. Data Padder accomplishes this job for each sequence which is fed by Parser.
4. Storing Spot history – In this step we store the parsed and padded data in a persistent storage to be used in price prediction step.
5. Reading Spot history from Storage – At the time of forecasting the Spot price, Spot history for a particular type of Spot instance is taken from the persistent storage.

6. Submit Spot history to the Spot predictor API – Spot predictor API is a Python implementation which uses LSTM model which is offered by TensorFlow's Keras's API for time series prediction. We call this API with the Spot Instance market price historical data.
7. Get LSTM predictions from Spot predictor API – Spot predictor API provides price prediction for a required time horizon by getting them trained in a LSTM model.
8. Get the workload prediction – In this step we use InteliScaler workload predictor without any amendments to get workload predictions for a given future time horizon because it has been proven to outperform all existing workload prediction models.
9. Get price predictions from Price Predictor module – Within the Optimal Bid Detector module, we apply the optimal bid selection procedure (see Section 5.2) to take the optimal bid for a given decision window.

5.2. Optimal Bid Selection Procedure

Using the InteliScaler's workload predictor and proposed novel LSTM model-based Spot predictor, we are able to closely predict workload and Spot price for a given time in future. The bid price is decided by looking at the Spot price fluctuations within a decision window. Here we choose the maximum possible Spot price within this interval to achieve the lowest possibility of that particular instance getting terminated by the provider whenever the market price goes beyond the value specified in Spot request.

For instance, Figure 5.2 shows how we applied above approach to find the bid value for making a Spot request when scaling decision is made for AWS t2 instance type. The upper diagram shows the anticipated workload fluctuation which has been 6generated by InteliScaler's workload predictor whereas the lower diagram shows the Spot price fluctuation of m4.4xlarge versus 2 times m4.2xlarge and the minimum fluctuation of both (for the simplicity we have only considered two instance types from the AWS t2 family). As the anticipated workload exceeds the scaling threshold, our approach considers a predefined time interval (i.e., decision window starting from that moment. Then it calculates the highest value of minimum fluctuation of both m4.4xlarge and two times m4.2xlarge fluctuations. This value can be used as the bid price for making Spot request and it can be assumed to be the best possible value which minimizes the chance out-of-bid occurrence. When the decision window is smaller the uncertainty is lower.

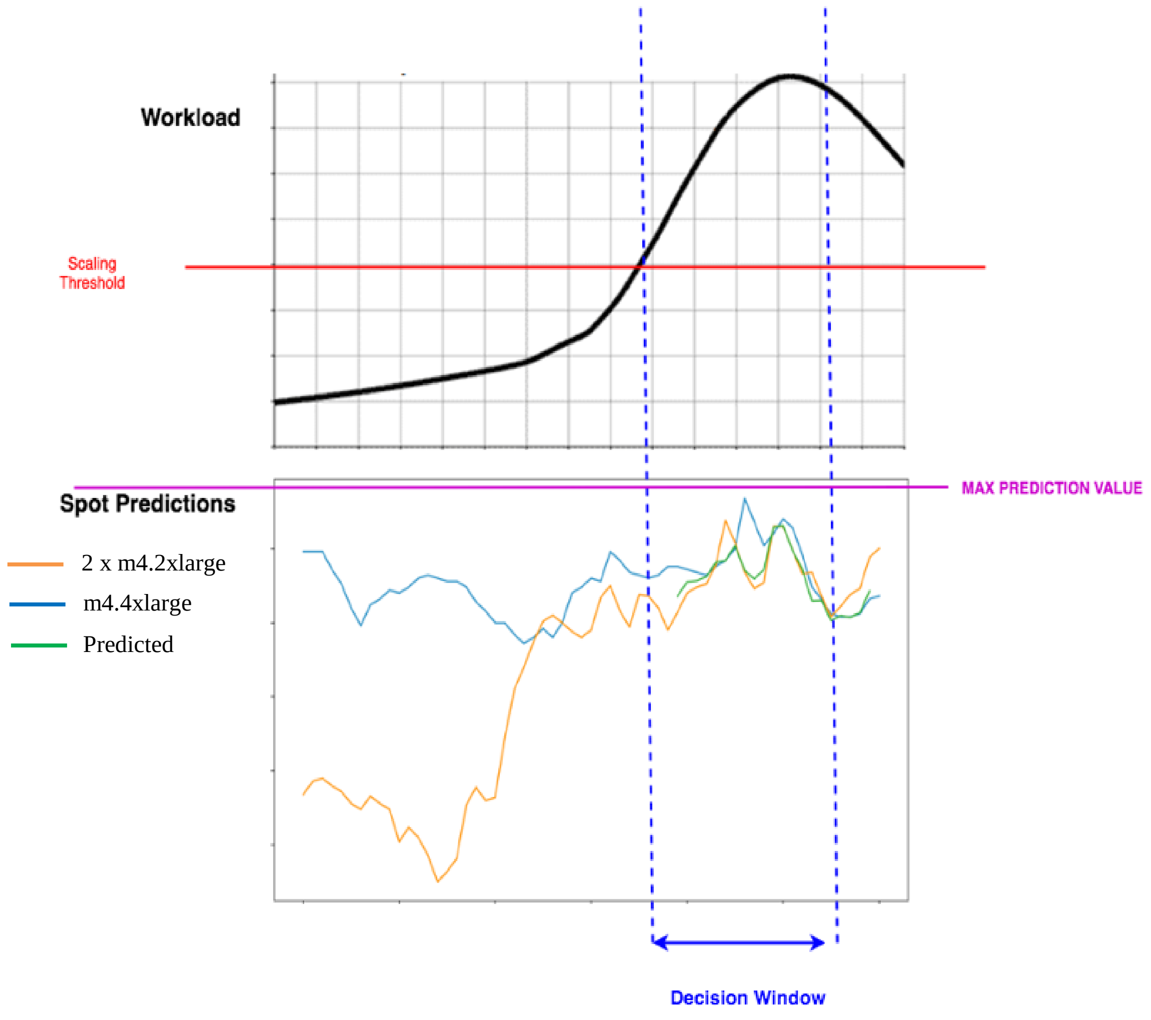


Figure 5.2: Decide the best bid value for m4.4xlarge & m4.2xlarge Spot instances when scaling decision is made.

5.3. Simulation-Based Evaluation

5.3.1. Simulation Setup

The entire setup consists of two major parts. In the first part scaling decisions are made using proactive auto scaling together with smart killing approach. In the second part we use the LSTM RNN predicted pricings to detect the Spot request value to be used when the scaling decision is made. According to the way that Spot price is predicted, we have taken results for two different simulations as following:

1. Use one Spot Instance type for scaling.
2. Use two different types of instances from the same family (i.e., r5) where one can be regarded as a single VM unit and a combination of other type equivalent to that unit.

5.3.1.1. Proactive Auto Scaling

In general, any auto-scaling solution which makes decisions based on the anticipated workload has an appropriate workload prediction mechanism at its backbone. For this simulation the ensemble prediction method mentioned in Section 4.1 is used to generate proactive auto scaling decisions. The penalty factor is calculated by using the following function.

$$f(x) = \begin{cases} 0, \ if\ 0 < x \le 0.05; \\ 0.1, \ if\ 0.05 < x \le 1; \\ 0.2, \ if\ 1 < x \le 5; \\ 2^{\frac{x}{20}}, \ if\ 5 < x \le 100; \end{cases} \tag{5.1}$$

In smart killing, an AWS VM is only terminated if it has been used beyond 50 minutes in previous billing cycle because AWS does not charge for a partial hour and we can utilize the full hour which it gets charged for. Considering the time for a graceful spin down, if the machine has been used beyond 57 minutes, it will not be shut down because it will have already charged for the next cycle as it actually gets physically spun down.

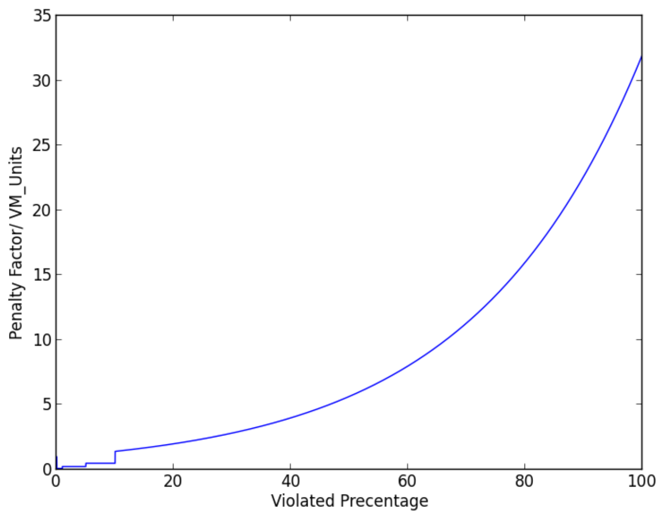
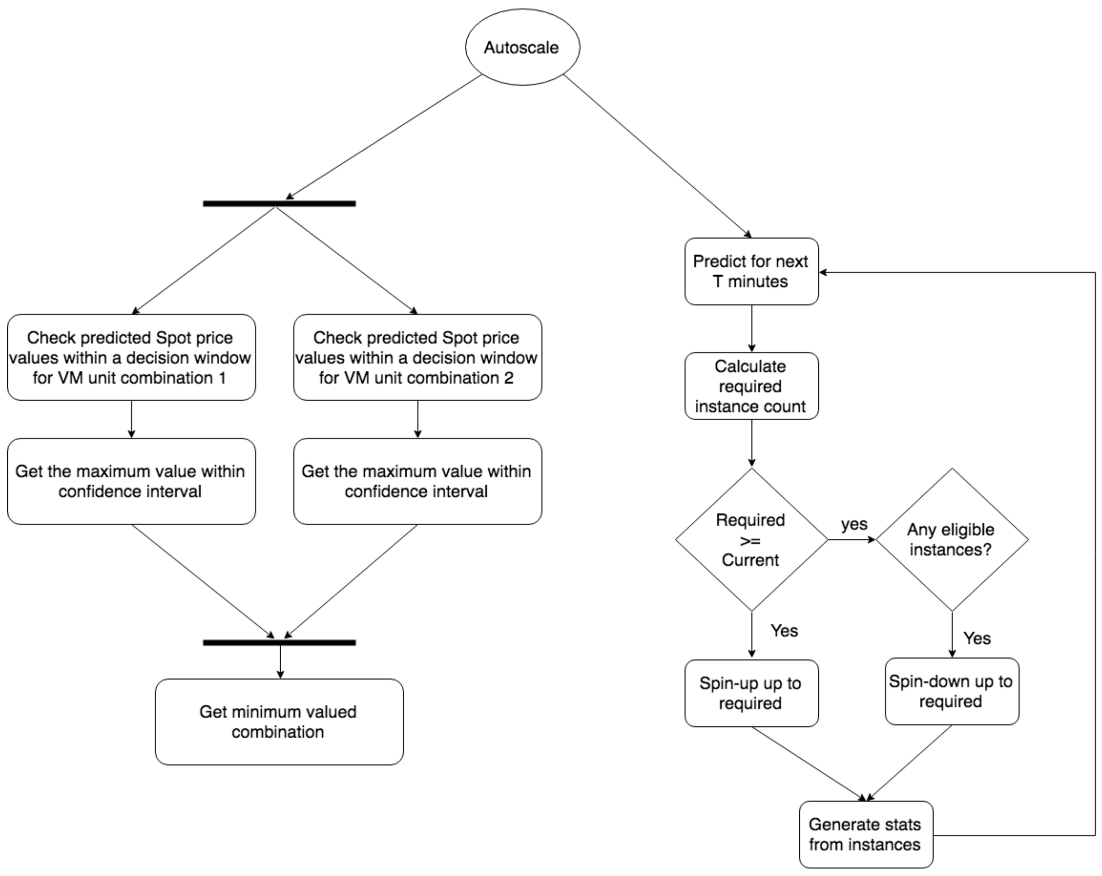


Figure 5.3: Penalty factor over violation percentage.



Autoscale
Check predicted Spot price values within a decision window for VM unit combination 1
Check predicted Spot price values within a decision window for VM unit combination 2
Get the maximum value within confidence interval
Get the maximum value within confidence interval
Get minimum valued combination
Predict for next T minutes
Calculate required instance count
Required >= Current
yes
Any eligible instances?
Yes
Yes
Spin-up up to required
Spin-down up to required
Generate stats from instances

Figure 5.4: Resource allocation.

5.3.2. Scaling Out with Homogeneous Units

Using above mentioned simulation setup we obtained the variation of VM count and load average overtime (i.e., 450 minutes) which is illustrated in Figure 5.5. In this first simulation we used only r5.2xlarge with 6-hour window for scaling out. Table 5.1 shows the violation cost, total cost, and violation percentage after 7-hours.

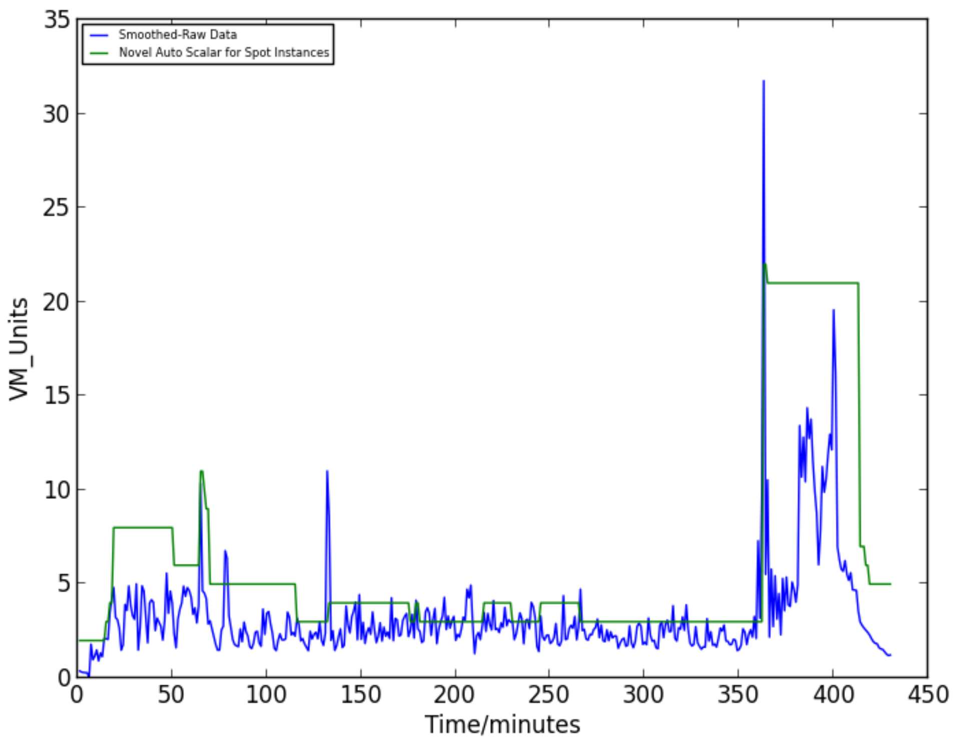


Figure 5.5: Variation of VM count and load average with time with the proposed approach for scaling out with homogeneous units (decision window = 6-hours).

Table 5.1: Violation cost, total cost, and violation percentage after 7-hours with the proposed approach for scaling out with homogeneous units (decision window = 6-hours).

Total Violation Cost	1.54416957
Total Cost	4.60544957
Violation Percentage after 7-hours	11.6822%

We also obtained the behavior of total cost throughout a considerable time period (450 minutes) for r5.2xlarge within 6-hour decision window. As shown in Figure 5.6, we compared it with the total costs for on-demand instances by using Reactive-Blind Killing, Stratos-Blind Killing, and Proactive Blind Killing. It can be clearly seen our proposed approach outperforms all above techniques in a very significant manner. By looking at its deviation from the graph of Reactive-Smart Killing clearly confirms its cost effectiveness as a result of using dynamically-priced instances.

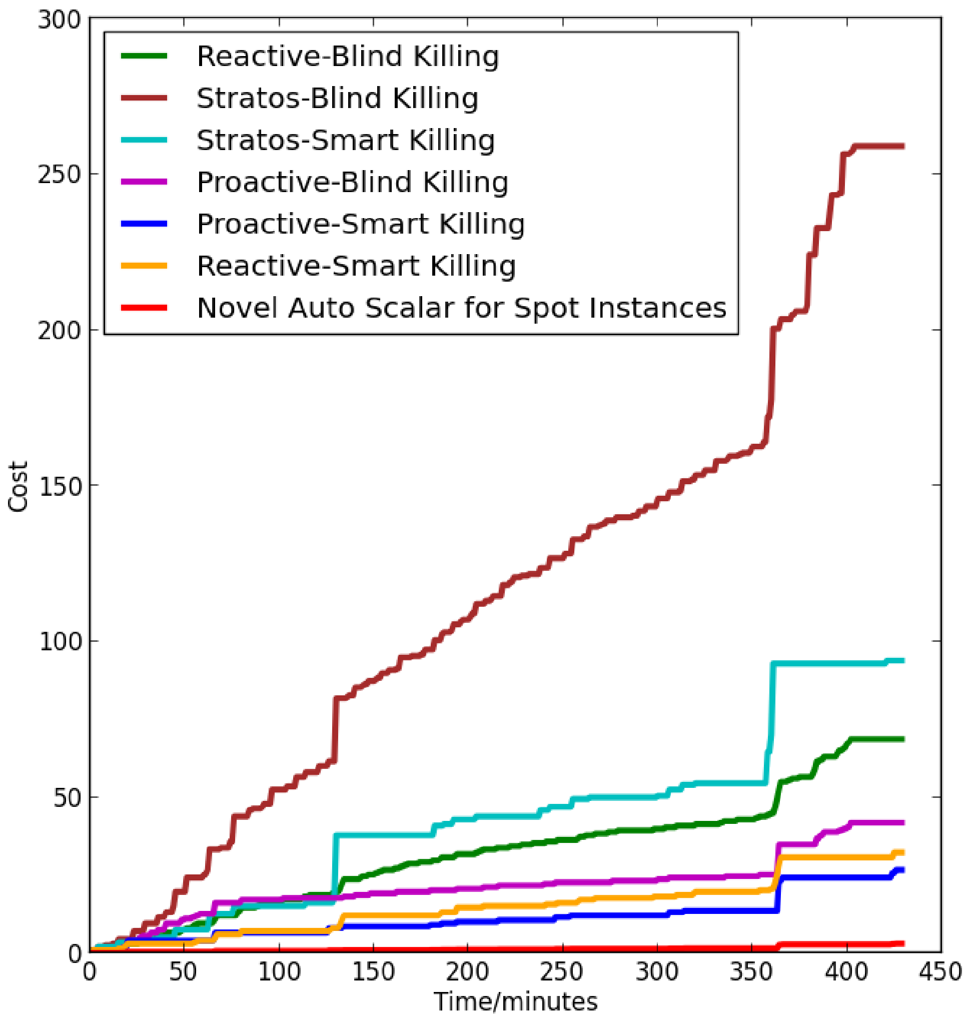


Figure 5.6: Variation of cost with time for different auto-scaling solutions when Spot decision window is 6-hours.

5.3.3. Scaling Out with Heterogeneous Units

Using the same simulation setup, we have obtained the fluctuation of VM count and load average overtime (i.e., 450 minutes) which we have shown in Figure 5.7. This time we have used both r5.2xlarge and r5.xlarge types with 6-hour decision window for scaling out. We have compared Spot request values of one r5.2xlarge instance versus two r5.xlarge instances and used the combination which has the minimum value for scaling. In the graph either r5.2xlarge instance or two r5.xlarge instances are regarded as a one VM unit. Table 5.2 shows the violation cost, total cost, and violation percentage after 7-hours.

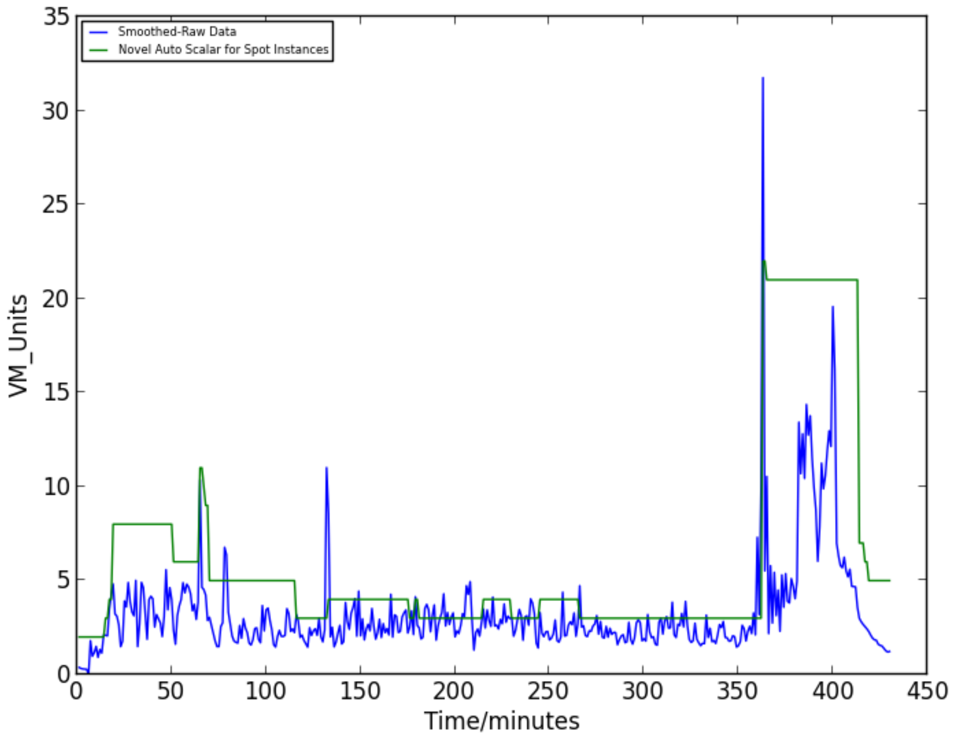


Figure 5.7: Variation of VM count and average load with time with the proposed approach for scaling out with heterogeneous units (decision window = 6 -hours).

We also obtained the behavior of total cost throughout a considerable time period (450 minutes) for r5.2xlarge with a 6-hour decision window. As shown in Figure 5.8 we also compared it with the total costs of on-demand instances while using Reactive-Blind Killing, Stratos-Blind Killing, and Proactive Blind Killing. Based on this comparison it can be clearly seen that the proposed approach outperforms all above techniques significantly. For instance, it reduces total cost by almost 75% when compared to Proactive Smart Killing. By looking at its deviation from the graph of Reactive-Smart Killing clearly confirms its cost effectiveness as a result of using dynamically-priced instances.

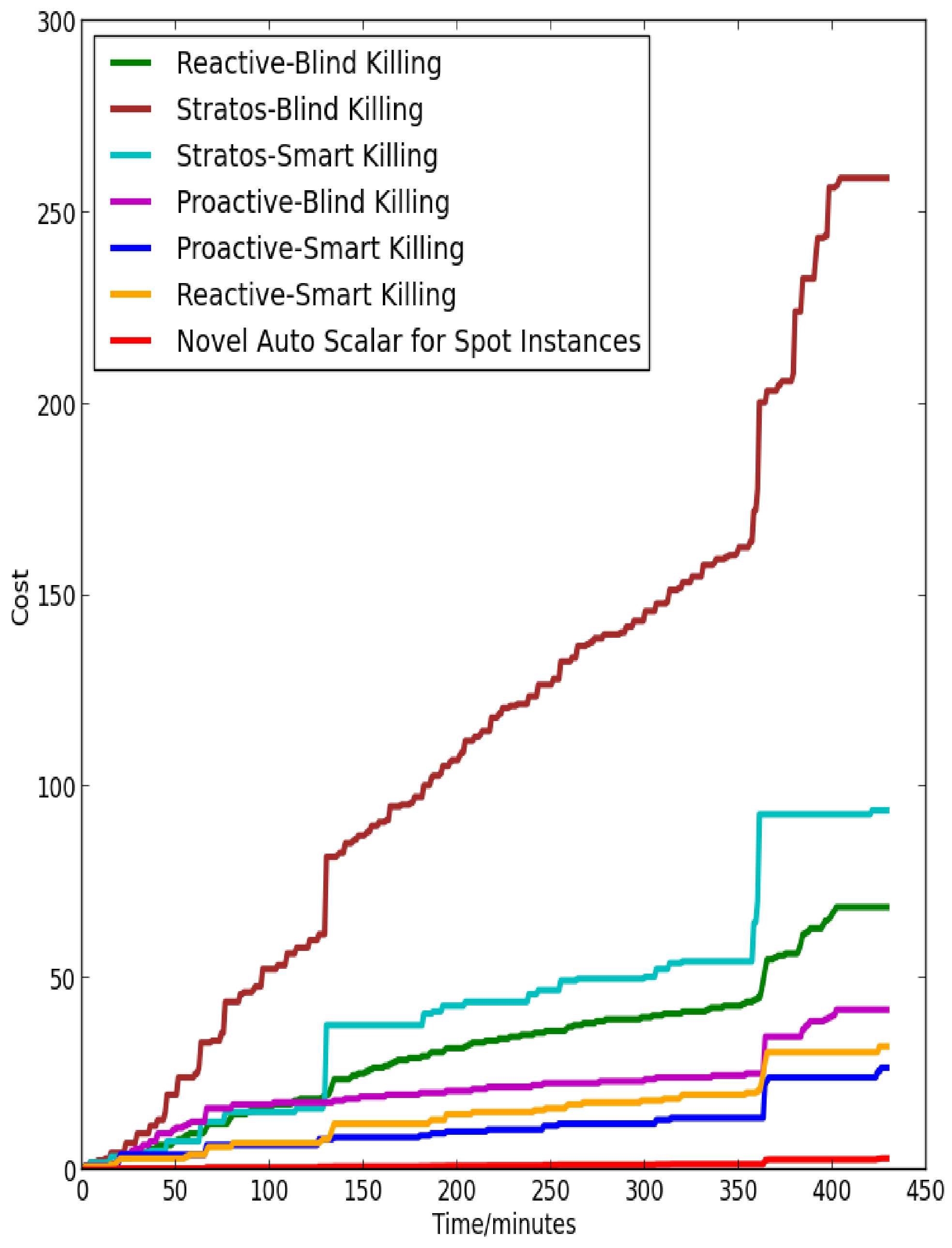


Figure 5.8: Variation of cost with time for different auto-scaling solutions when Spot decision window is 6-hours.

Table 5.2: Violation cost, total cost, and violation percentage after 7-hours with proposed approach for scaling out with heterogeneous units (decision window = 6-hours).

Total Violation Cost	1.23533565565
Total Cost	4.29661566
Violation Percentage after 7-hours	11.6822%

Table 5.3: Violation Cost, Total Cost and Violation Percentage after 7-hours for different auto-scaling methods

	Total Violation Cost	**Total Cost**	**Violation Percentage after 7-hours**
Reactive Smart Killing	3.121386	35.441386	32.0093%
Proactive Smart Killing	1.543100	28.30810	11.6822%
Novel Method for Spot Instances(one instance type)	1.544169	4.605449	11.6822%
Novel Method for Spot Instances (best of 2 combinations from same instance family)	1.235336	4.296616	11.6822%

From Table 5.3 it can be seen how different setups (i.e., homogeneous and heterogeneous) outperforms both reactive and proactive smart killing methods. At the same time when comparing the total violation cost and total cost of both homogeneous and heterogeneous setups, it is clear that the heterogeneous setup is more beneficial. Therefore, for further analysis we focus on heterogeneous setup.

5.4. Empirical Evaluation

5.4.1. Experimental Setup

To accomplish a quantitative and complex comparison of the quality of service between InteliScaler and our novel auto scaler for Spot instances, we came up with following two simulation setups depicted in Figure 5.9 and 5.10. These setups mimic the core behaviors of InteliScaler, as well as our proposed solution for Spot instances respectively. We have used them to get various measurements about the quality of service of the proposed solution. Even though the InteliScaler has been experimented on Apache Stratos, we decided came up with these solutions as Stratos does not inherently support EC2 instances and if we are to introduce that, it will be a huge architectural change for it. The only difference is; in these simulation setups we do not use the Startos' exact way of getting monitoring metrics and provisioning instances. However, we used InteliScaler Workload Predictor Model and Cost Calculation Model (i.e., the model which decides on getting scaling decisions) as themselves.

In the simulation setup, InteliScaler consisted of five major modules which are Web-socket API, AWS EC2 Requester Service, ELB Register/Deregister Service, Scale Decision Service and Workload Store. The Web-socket API can get workload matrices (i.e., CPU usage) through web socket connections between Spot Instances and itself. The socket is created by an agent who is placed inside each instance and it automatically starts at the creation of On-Demand Instance. Workload metrics will be averaged and stored in workload store which follows a sliding-window-based mechanism to get latest workload metrics by shifting out the older values.

Scale Decision Service checks within constant time intervals whether it should take any scaling decision. This is done by extracting the workload values from workload store and submitting them to an R server which includes InteliScaler ensemble method workloads to get workload predictions. Then the service submits those predictions to the Cost Model to check whether any scaling decision to be taken.

AWS EC2 Requester Service is capable of starting EC2 on-demand instances whenever it is needed. It also uses ELB register/deregister service to register services with ELB at the On-Demand Instance creation time, as well as deregister them at their terminations. On-demand instances are spin up using an image which consists of RUBiS webserver and agent which is initiated via an upstart job.

Here we have used HA Proxy to measure quality of service measurements like requests in flight and requests queued. We had to use an Elastic Load Balancer at the middle of HA Proxy and on-demand instances because it is impossible to register target service with HA Proxy from a remote location. However, ELB could do it through AWS APIs via an AWS client.

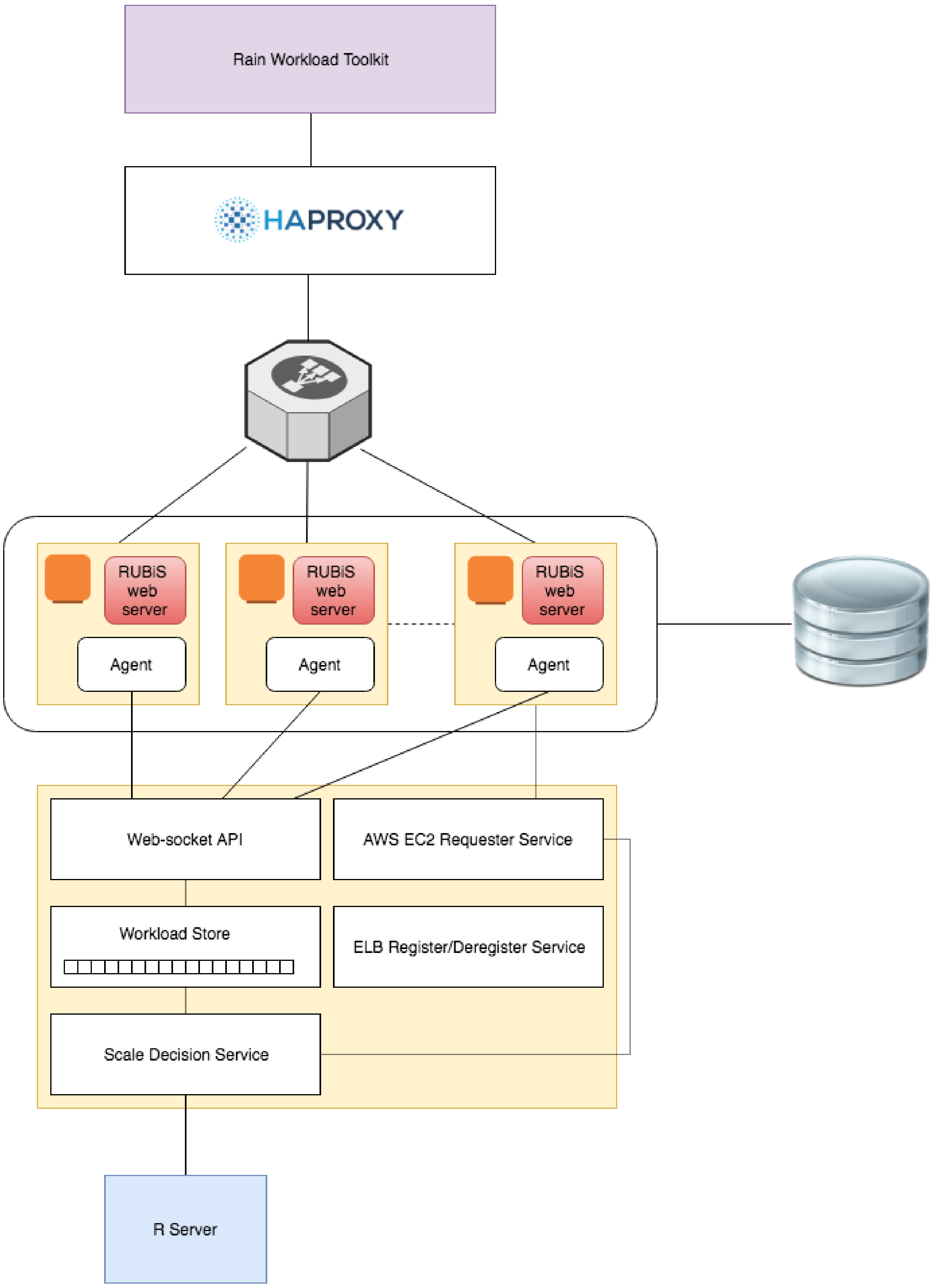


Figure 5.9: Simulation setup that mimic InteliScaler.

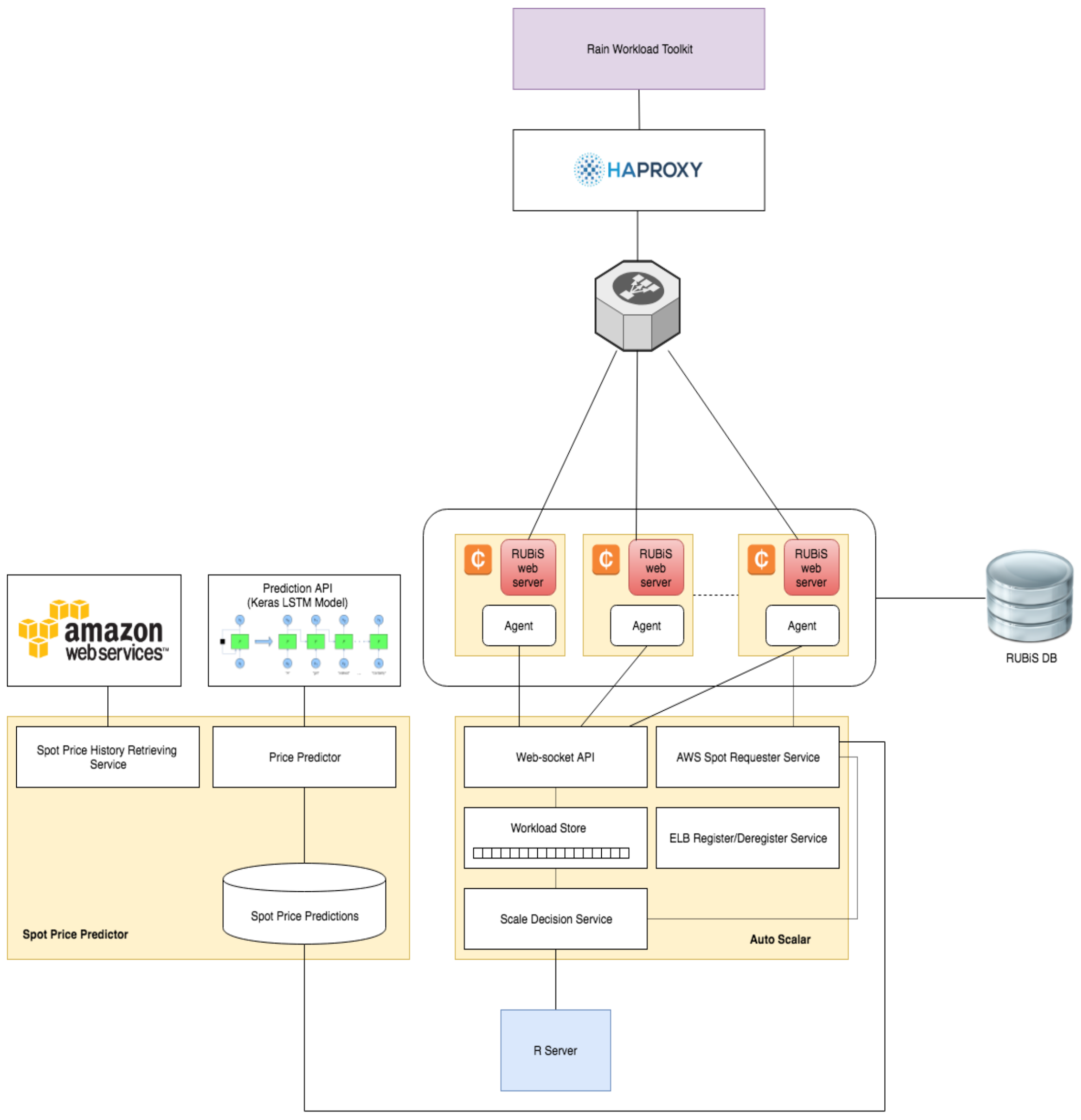


Figure 5.10: Simulation setup that mimic novel auto scaler for Spot instances

The simulation setup for price and workload aware auto scaler for Spot instances consists of two major components. They are the Price Predictor and Auto scaler. The Price Predictor consumes AWS's special API to get Spot price history and get predictions by submitting them into a Prediction API based on Keras LSTM Model which is hosted remotely (as it requires lot of CPU intensive operations). Price Predictor ultimately stores all predictions for the usage of Auto scaler.

Auto scaler consists of five major modules which are Web-socket API, AWS Requester Service, ELB Register/Deregister Service, Scale Decision Service and Workload Store.

The Web-socket API is capable of getting workload matrices (in this case CPU usage) through web socket connections between Spot Instances and itself. The socket is created by an agent who is placed inside each instance and it automatically starts at the creation of Spot instance. Workload matrices will be averaged and stored in workload store which follows actually a sliding window-based mechanism to get latest workload metrics by shifting out the older values.

The Scale Decision Service checks within constant time intervals whether it should take any scaling decision. This is done by extracting the workload values from workload store and submitting them to an R server which includes InteliScaler ensemble method workloads to get workload predictions. Then the Scale Decision Service submits those predictions to the Cost Model within it to check whether any scaling decision to be taken.

AWS Spot Requester Service is capable of kicking off Spot Requests based on the price suggested from the Price Predictor Component and terminate an instance whenever needed. It also uses ELB register/deregister service to register services with ELB at the Spot instance creation time, as well as deregister them at their terminations. Spot Instances are spin up using an image which consists of RUBiS webserver and agent which is initiated via an upstart job.

Here we have used HA Proxy to measure quality of service measurements like requests in flight and requests queued. We had to use an Elastic Load Balancer at the middle of HA Proxy and Spot Instances because it is impossible to register target service with HA Proxy from a remote location. However, ELB has that capability as it can be done through AWS APIs by using an AWS client.

Here, the synthetic workload has been produced by a toolkit for workload generation called RAIN which was developed by University of California at Berkeley [17] which uses famous RUBiS auction site as the archetype [18]. IntelliScaler research provides sufficient justifications for using RAIN generator to mimic the RUBiS site user's behavior as a better candidate. It further explains how RAIN kit addresses scaling and low flexibility issues while simulating clients. Therefore we have chosen that approach straightaway. As our target RUBiS web server is written in PHP language we selected the PHP in flavor of the RUBiS setup in our simulations. Apache webserver is used to serve RUBiS and a MySQL database server is used in addition. RAIN simulator runs on a Java runtime.

RUBiS is hosted on top of Apache 2.4.7 web server, complemented by MySQL 5.5.44 database server. Rain workload generator is run on top of Java 1.7 OpenJDK runtime environment.

5.4.2. Quality of Service Measurements

Using the Rain workload generator, we have generated the following workload profile on two simulation setups. We have taken the same quality of service measurement three times in each setup to avoid if there are any outlying conditions.

Table 5.4: Workload profile for simulation set-up.

Interval(sec)	90	270	90	90	90
No of Users	400	1500	1700	800	800
Ramp up time (sec)	10	10	10	10	10

Table 5.5: Results in the simulation setup for mimic InteliScaler.

	Iteration 1	**Iteration 2**	**Iteration 3**
Total Operations	99561	100313	100171
Failures	34833	65019	63676
Response Time	0.2101	0.2284	0.2060
Operations Late	10	12	12
Error Rate	34.99%	64.81%	63.56%
Read Timeouts	259	10	444

As per the results in Table 5.6 failure rate varied with a lower bound of 34.99% and an upper bound of 64.81% in simulation setup for mimic InteliScaler. Also, as per the results in Table 5.10 failure rate varied with a lower bound of 35.06% and an upper bound of 63.63% in simulation setup for novel price and workload-aware auto scaler for Spot instances. Therefore, it is very clear that our novel method does not show any exceptional behavior with regards to Failure rate.

Table 5.6: Scaling Decisions – Iteration 1.

Action	**Time**
Start ec2 instance	179
ELB registration	211
Instance removal	497
Start ec2 instance	529
ELB registration	588
Instance removal	655

Table 5.7: Scaling Decisions – Iteration 2.

Action	**Time(s)**
Initiate Spot Request	149
ELB registration	181
Instance removal	250
Initiate Spot Request	279
ELB registration	310
Instance removal	500

Table 5.8: Scaling Decisions – Iteration 3.

Action	**Time(s)**
Initiate Spot Request	174
ELB registration	206
Instance removal	270
Initiate Spot Request	303
ELB registration	336
Instance removal	399
Initiate Spot Request	462

ELB registration	495
Instance removal	622

Read Timeouts of simulation setup for mimic InteliScaler have a lower bound of 10 and upper bound of 444 as seen in Table 5.6. In the novel price and workload-aware auto scaler for Spot instances we varied the read timeouts from 100 to 430. Therefore, it is clear, even in this context our novel method does not show significant deviation from the same figures of simulation setup for mimicking InteliScaler.

Table 5.9: Results of novel price and workload aware auto scaler for Spot instances.

	Iteration 1	**Iteration 2**	**Iteration 3**
Total Operations	100476	100264	101043
Failures	35230	36841	64295
Response Time	0.2138	0.2088	0.1975
Operations Late	21	14	4
Error Rate	35.0631%	36.74%	63.63%
Read Timeouts	148	430	100

Table 5.10: Scaling Decisions – Iteration 1.

Action	**Time**
Initiate Spot Request	142
ELB registration	231
Instance removal	516
Initiate Spot Request	549
ELB registration	589
Instance removal	623

Table 5.11: Scaling Decisions – Iteration 2.

Action	**Time(s)**
Initiate Spot Request	163
ELB registration	216
Instance removal	242
Initiate Spot Request	280
ELB registration	321

Instance removal	384

As a summary, when comparing the general measurements taken by InteliScaler and our novel auto-scaler for Spot instances (see Figure 5.5 and 5.9), it is clear that no any effect has happened to the quality of service when using Spot instances rather than using on-demand instances. Even though we can see some fluctuation of error rate in both scenarios, it can be seen that the response time remains in the same region. Also, by looking at scaling decisions taken in both scenarios (Table 5.6 to Table 5.12) we can conclude that time taken to spin up and instance is almost the same and does not affect the quality of service.

Table 5.12: Scaling Decisions – Iteration 3.

Action	**Time(s)**
Initiate Spot Request	147
ELB registration	188
Instance removal	221
Initiate Spot Request	254
ELB registration	298
Instance removal	366
Initiate Spot Request	393
ELB registration	421
Instance removal	485

Figure 5.11 shows how the average response time varies when calling to different endpoints of RUBiS web application. Out of many endpoints of RUBiS web application here we chose /Register endpoint, /Brower endpoint, and /Sell endpoint. Based on these figures it is clear that the proposed auto-scaler for dynamically-prices VMs outperforms even InteliScaler's equivalent solution in most occasions.

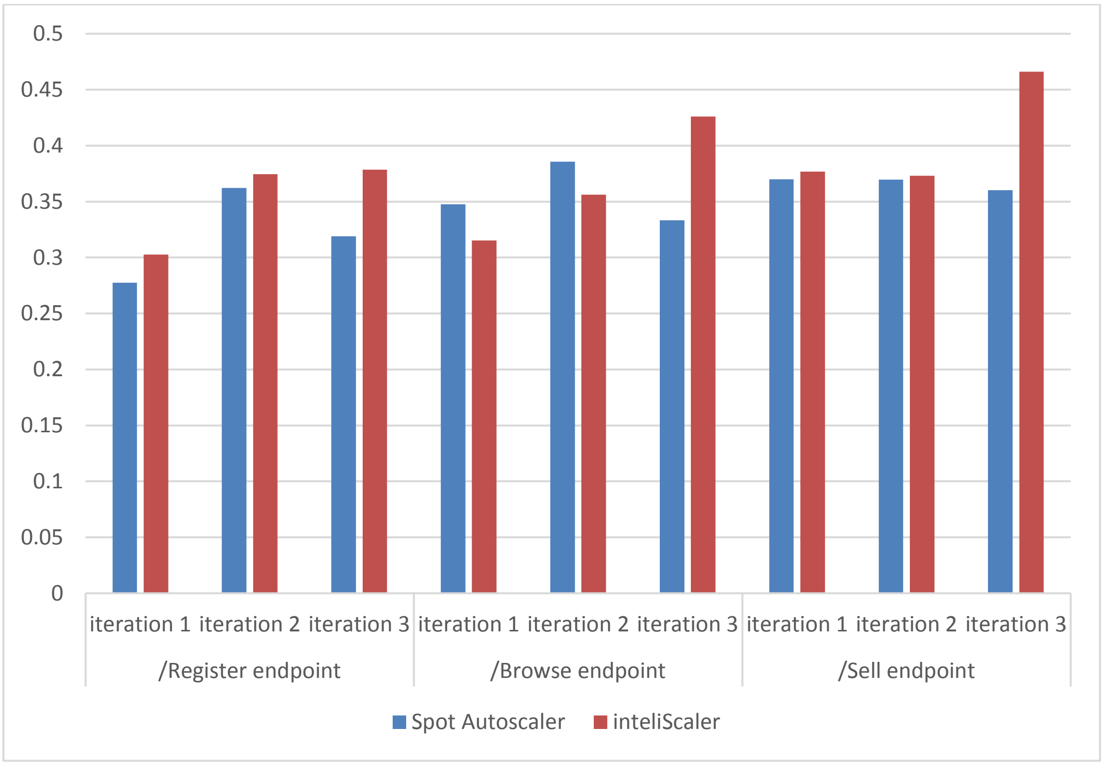


Figure 5.11: Average response time of selected endpoints of RUBiS web application.

6. Summary

Section 6.1 presents the conclusion. Research limitations are presented in Section 6.2. Section 6.3 presents future work.

6.1. Conclusion

Auto-scaling is capable of provisioning compute resources to a cloud application. Hence, it is one of the major facts which optimize the resource utilization of them. is Achieving high QoS (Quality of Service) measures while minimizing the associated costs is the ultimate target of auto-scaling [1].

Being aware of the workload that a particular system experiences and intelligent provisioning of capacities in the most profitable manner focusing keenly on Quality of Service are the key challenges of an auto-scaling system.

In the context of cloud computing, the first challenge is addressed with two different approaches. One approach is gaining the workload awareness for auto scaling in a reactive manner. The other approach is gaining the workload awareness for auto scaling proactively. Auto scaling decisions are being taken based on a predefined set of events and guidelines in the reactive approach. In contrast, the proactive approach makes decisions based on anticipated workload by forecasting the workload ahead of time, rather than waiting for event triggers. However, the major challenge of forecasting workload is coming up with a mechanism which can be used to make predictions for various workload patterns. For instance, a particular model that performs well on seasonal trends may not perform well with frequently fluctuating loads.

Recently when people choosing auto-scaling methods, they tend to select the proactive approach rather than reactive approach because proactive way uses forecasting in backbone which provides auto-scaler to come up with fine-grained decisions. From our perspective, the reactive approach is much simpler, but when it comes to resource allocation it greatly shrinks the solution space. With the novel approach called InteliScaler researchers have proposed a proactive auto-scaling method which is capable of make workload predictions ahead of time using an ensemble technique of combining time series forecasting, as well as machine learning techniques. In terms of accuracy the proposed ensemble method outperforms individual techniques as per the results.

However, InteliScaler and all the other methods mentioned above have focused on AWS On-Demand Instance type as the candidate instance type to apply proactive auto scaling. Recently AWS released another type of instances called Spot Instances which have a dynamic pricing model and are based on user bids. Through Spot Instances, AWS is able to sell their spare capacity of datacenters. Spot Instances or variable pricing Virtual Machines (VMs) are widely used today to save cost. Thus, it is a real deficiency if we do not extend the capabilities of auto scalers to cater Spot Instances as well. To address this concern, we proposed a novel workload and resource-aware auto scaler for Spot Instances. We have implemented this approach as an extension to the InteliScaler which reuses its workload predicting model and Recurrent Neural Networks (RNN) based model for predicting Spot Instance prices. For making the decision on the number of resources required this novel method considers cost as well as QoS factors. We tested the performance of this approach by comparing the total cost of using it with the total cost of other auto-scaling methods which had been proposed for AWS on-demand instances. Further, we have shown how the total cost of our approach reduces when we decrease the interval that we need to have a Spot Instance is up and running.

According to our performance evaluation total cost of our proposed approach has shown 75% reduction compared to InteliScaler which caters On-Demand Instances. Moreover, we compared the quality of service of our proposed approach with InteliScaler using a famous benchmark technique. It proved that there is no performance degradation associated with our novel auto-scaler for Spot Instances with respect to InteliScaler.

6.2. Research Limitations

There were many challenges to be addressed in this research since AWS only provides Spot price history which is only three months old. Hence, this data set cannot be regarded as a best candidate since we miss various other fluctuation patterns during some time periods of a given year. For instance, in December we may expect very different fluctuation in Spot price due to Christmas and New Year. Here we had to artificially generate such data sets.

During the price prediction process, we had to use data which is separated from uniform time gaps. However, AWS provides data which is corresponding to the fluctuation points. Therefore, we had to add padding values to the missing points to feed them as input values to ARIMA and LSTM like time-series-based prediction models. Anyway, we didn't need any interpolation when deciding padding values because we can assume Spot price is a constant from one fluctuation point to the other point.

During the quality of service test which we have done at last also has several limitations. Up to which extent that we can compare quality of service of a setup hasn't auto-scaling with a setup which has auto scaling is doubtful. Even though we expected a reduction in failure rate by using a setup with auto-scaling, it didn't happen because various problems of target application (here RUBiS) related to session management can cause different failures. Therefore, we believe if we have used two tier application which has been well improved its scalability, we could have received more positive results.

6.3. Future Work

In the provided solution, and the proposed auto-scaling implementation of proactive auto-scaler, InteliScaler assumes the homogeneity of the worker nodes (i.e., instances). Even though we mentioned in the proposed algorithm that we should consider about all Spot Instance permutations from the same family to get the best possible bid value for a given decision window, we considered only one EC2 instance type for the experimental setup. While considering all possible permutations of Spot Instances at the time when scaling decisions are taken is not straight forward, a detailed performance analysis is needed to determine their impact.

The proposed auto-scaler for dynamically priced VMs (in this case AWS Spot Instances) focused on limited set of instances, as cost factor did not permit us to extend this research throughout all instance types that AWS offers. To complete the proposed solution, a proper mapping is required between the workload requirement and the suitable configuration of instance type which is available at the IaaS layer.

While we decide our price based on a decision window and select the maximum possible value as the bid price, there is a lower chance of experiencing out of bidding situation. However, due to various types of exceptions there is a probability of Cloud provider terminating the Spot Instance spin up by our novel auto-scaler. To cater such scenarios, we need to give our auto-scaler the capability of preserving the state in failure situations. This can be done either by using various existing methods like checkpointing or any other approach for saving the instance state in out-of-bid situation.

References

[1] R. Shariffdeen, D. Munasinghe, H. Bhathiya, U. Bandara, and H.M.N.D. Bandara, “Workload and Resource Aware Proactive Auto-scaler for PaaS Cloud,” in Proc. *19th IEEE Intl. Conf. on Cloud Computing (CLOUD),* 2016.

[2] S. Tang, J. Yuan, and X.Y. Li, “Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance,” *2012 IEEE 5th Intl. Conf. on Cloud Computing*, 2012.

[3] W. Voorsluys and R. Buyya, “Reliable Provisioning of Spot Instances for Compute-intensive Applications,” *2012 IEEE 26th Intl. Conf. on Advanced Information Networking and Applications,* 2012.

[4] S. Yi, D. Kondo, and A. Andrzejak, “Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud,” in Proc. *2010 IEEE 3rd International Conference on Cloud Computing,* 2010.

[5] S. Yi, J. Heo, Y. Cho, and J. Hong, “Taking point decision mechanism for page-level incremental checkpointing based on cost analysis of process execution time,” *Journal of Information Science and Engineering,* vol. 23, no. 5, pp. 1325–1337, Sep. 2007.

[6] WSO2 Private PaaS. [Online]. Available: http://wso2.com/cloud/private- paas

[7] Amazon Web Services, Inc.(2017). *Amazon EC2 Spot Instances*. [Online]. Available: http://aws.amazon.com/ec2/spot-instances/

[8] M. Mazzucco and M. Dumas, “Achieving performance and availability guarantees with Spot instances,” in *13th Intl. Conf. on High Performance Computing and Communications (HPCC).* Los Alamitos, CA, USA: IEEE Comput. Soc., 2011.

[9] Amazon Web Services, Inc.(2017). *Amazon EC2 Pricing* [Online]. Available:https://aws.amazon.com/ec2/pricing/

[10] C. Bunch, V. Arora, N. Chohan, C. Krintz, S. Hegde, and A. Srivastava, “A pluggable autoscaling service for open cloud PaaS systems,” in *Proc. 5th IEEE Intl. Conf. on Utility and Cloud Computing,* Nov. 2012. [Online]. Available: http://dx.doi.org/10.1109/ucc.2012.12

[11] Kim Weins (2016, Nov. 28). AWS vs Azure vs Google Cloud Pricing: Compute Instances. [Online]. Available: https://www.rightscale.com/blog/cloud-cost-analysis/aws-vs-azure-vs-google-cloud-pricing-compute-instances

[12] Amazon Web Services, Inc.(2018). *describe-spot-price-history* [Online].

Available: https://docs.aws.amazon.com/cli/latest/reference/ec2/describe-spot-price-history.html

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

[14] Unknown (2015). *Understanding LSTM Networks* [Online].

Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[15] 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.

[16] R. Shariffdeen, D. Munasinghe, H. Bhathiya and U. Bandara, (2015) *AutoscaleAnalyser* [Online]. Available:https://github.com/hsbhathiya/AutoscaleAnalyser

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

[18] OW2 Consortium (2015). Understanding LSTM Networks [Online].

Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/