CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **28**:3672–3690
Published online 23 December 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3749

SipaaS: Spot instance pricing as a Service framework and its implementation in OpenStack

Adel Nadjaran Toosi*,†, Farzad Khodadadi and Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne Melbourne, Australia

SUMMARY

Designing dynamic pricing mechanisms that efficiently price resources in line with a provider's profit maximization goal is a key challenge in cloud computing environments. Despite the large volume of research published on this topic, there is no publicly available software system implementing dynamic pricing for Infrastructure as a Service cloud spot markets. This paper presents the implementation of a framework called *Spot instance pricing as a Service* (SipaaS) that supports an auction mechanism to price and allocate virtual machine instances. SipaaS is an open-source project offering a set of web services to price and sell virtual machine instances in a spot market resembling the Amazon EC2 spot instances. Cloud providers, who aim at utilizing SipaaS, should install add-ons in their existing platform to make use of the framework. As an instance, we provide an extension to the *Horizon* – the OpenStack dashboard project – to employ SipaaS web services and to add a spot market environment to OpenStack. To validate and evaluate the system, we conducted an experimental study with a group of 10 users utilizing the provided spot market in a real environment. Results show that the system performs reliably in a practical test environment. Copyright © 2015 John Wiley & Sons, Ltd.

Received 23 August 2015; Revised 20 November 2015; Accepted 20 November 2015

KEY WORDS: cloud computing; market; dynamic pricing; auction; spot instances; OpenStack; framework

1. INTRODUCTION

Infrastructure as a Service (IaaS) cloud providers have started offering unused computational resources in the form of dynamically priced virtual machines (VM instances) [1]. Dynamic pricing is a pricing strategy in which providers set prices for their services based on current market demands. The fact that demand for cloud services and computational resources is non-uniform over time motivates the use of dynamic forms of pricing in order to optimize revenue. Hence, design and implementation of dynamic pricing mechanisms have received considerable attention in the literature [2–4].

There are various types of dynamic pricing strategies suggested by the literature, for example, different types of auctions [5, 6], negotiations [7], and yield management techniques [3]. Auction is among the most popular techniques, which is a common market mechanism with a set of rules determining prices and resource allocations on basis of bids submitted from the market participants. Amazon Web Services (AWS)‡ is one of the pioneers who sell spare capacity of data centers using an auction-like dynamic pricing mechanism. In Amazon terminology, VM instances trading in this form of pricing is known as *spot instances*, and the market in which spot instances are traded is called *spot market*.

*Correspondence to: Adel Nadjaran Toosi, CLOUDS Lab., Department of Computing and Information Systems, The University of Melbourne, Australia.
†E-mail: adel.nadjaran@unimelb.edu.au
‡Amazon Web Services (AWS), http://aws.amazon.com.

Spot market, since introduced by AWS, has been considered as one of the first steps towards a full-fledged market economy for computational resources [8]. In the spot market, customers communicate their bids for an instance-hour to AWS in order to acquire required number of instances. Subsequently, AWS reports a market-wide *spot price* at which VM instance usage is charged while terminating instances that are executing on a bid price lower than the market price (*out-of-bid* situation). Prices vary independently for each instance type and available data center (or *availability zone* in Amazon terminology).

Amazon Web Services has revealed no detailed information regarding their pricing mechanism and the computation of the spot price. At present, the design of dynamic forms of pricing for cloud computing resources is an open research challenge and of great interest to both cloud providers and researchers. An auction mechanism is truthful, if for each bidder irrespective of any choice of bid by all other bidders, the dominant strategy is to report his/her true information. We presented a pricing mechanism called *Online Extended Consensus Revenue Estimate* (online Ex-CORE) auction that is *truthful* with high probability and generates a near optimal profit for the cloud provider in [9].

In this paper, we describe an open source framework called *Spot instance pricing as a Service* (SipaaS). SipaaS[§] offers a set of web services that can be used by IaaS cloud providers to run a spot market resembling the AWS spot instances. It provides services to price VM instances using the internal pricing module that works based on the online Ex-CORE auction mechanism. The extensible architecture of the SipaaS framework allows for implementation of any new pricing mechanism without the necessity to modify the design of the web services. The purpose of the SipaaS framework is twofold: (1) providing a fully operational open-source software that can be used by IaaS cloud providers to offer a cloud spot market; and (2) providing an extensible software framework for conducting research on dynamic pricing techniques. Using the SipaaS framework, research community are able to plug their own developed pricing mechanisms into a practical environment and test scenarios in a spot market with known pricing module rather than only analysing existing commercial systems such as AWS spot instances without being aware of its pricing mechanism.

Cloud providers, who aim at utilizing SipaaS, require to extend their platform to make use of the framework. In this paper, we provide an extension to the OpenStack project[¶] as an example to utilize SipaaS web services. To facilitate research efforts and future advancements in the area of dynamic pricing for cloud spot markets, this paper outlines an extension to the *Horizon – the OpenStack dashboard project* – to make use of SipaaS and create a spot market environment for the OpenStack platform.

To validate and evaluate the system consisting of the SipaaS framework combined with the extension to OpenStack, we conducted an experimental study with a group of ten participants utilizing the provided spot market. Results show that the system is fully operational in a practical test environment and confirm the theoretically proven and using simulation shown truthfulness feature of the Ex-CORE auction.

The remainder of this paper is organized as follows: Section 2 discusses some background and related work. Section 3 confers the system design and implementation where we describe SipaaS, extensions to Horizon, and Ex-CORE pricing mechanism in subsections 3.1, 3.2, and 3.3, respectively. Evaluation and validation of the system is conducted in Section 4. And finally, conclusions are presented in Section 5.

2. BACKGROUND AND RELATED WORK

Pricing is the process of determining the rate or fee the provider will receive in exchange for offering services or selling resources. Cloud providers can use a variety of pricing strategies when selling their services. Among these strategies, *dynamic pricing* is a time-based and price discrimination scheme that allows the service provider to vary the price in real-time in response to various factors such as *market demands*, *the time of service offering*, *the type of customer*, and *the type of resources*

[§]SipaaS in Persian language means thank.
[¶]OpenStack: An open source software for building private and public clouds, http://www.openstack.org/.

or services. In general, dynamic pricing can be determined as [10]: a provider revenue maximization problem in a monopoly market, for example [3, 6, 11, 12], or a social welfare maximization problem in a competitive market with multiple providers, studies such as [13–16]. In this paper, we study the former and propose a framework that can be used by IaaS cloud providers to setup a dynamic pricing-equipped spot market.

There are various types of dynamic pricing strategies suggested by the literature for setting the price of cloud resources. These dynamic pricing strategies can be categorized into two main groups: *price-discovery* and *price-posted* models [17]. In the former, the provider sets the price based on the communication with the customers, for example, asking them to report their bid. *Auction-based* and *negotiation-based* [7, 18] techniques fall into this group. The latter, the price-posted model, does not necessarily require communication with the customers, and the provider posts the pre-determined price, which dynamically varies during the time based on some external factors such as demand or time of use [3]. The demand-oriented pricing model [19, 20] and yield management techniques [3, 21, 22] are categorized in the second group. The spot pricing framework proposed in this paper is specifically designed for auction-based techniques where customers submit their bid to acquire resources.

Over the recent years, there has been a massive growth in the research of designing auctions, largely motivated by the development of the Internet. Auctions can be in assorted shapes and have different characteristics such as: *single-dimensional* (e.g., only bid price) or *multi-dimensional* (e.g., bid price plus quantity), *single-sided* (e.g., only customers submit bids) or *double-sided* (e.g., both providers and customers submit bids), *open-cry* or *sealed-bid*, *single-unit* (e.g., a single good or service) or *multi-unit* (e.g., multiple units of the goods), *single item* (e.g., one type of service) or *multi-item* (e.g., combinatorial auction). These have been extensively discussed and analyzed in the economics literature. Interested readers are referred to [23] for a general survey on auction mechanisms from a computer science perspective. Apart from all the different types of auction that can be devised, auction designer might have specific goals in designing auction, for example, *truthfulness*, *revenue maximization*, *allocative efficiency*, and *fairness* [17]. This work focuses on the design and implementation of a software system where dynamic pricing algorithms working based on customers' bids, can be plugged into it to create a cloud spot market. Here, we use our previously proposed auction-based pricing mechanism called Ex-CORE [9], aiming at revenue maximization, truthfulness, and fairness.

There are relevant and similar auction mechanisms in the literature that can be similarly used in the pricing module of the proposed software system. Wang *et al*. [6] designed near-optimal dynamic auctions scheme to determine how many spot instances must be auctioned-off in each auction period to maximize the seller's revenue. Same authors in [2] have proposed an optimal recurrent auction for a spot market based on the seminal work of Myerson [24]. The mechanism was designed in the context of optimally segmenting the provider's data center capacity between on-demand and spot market requests. They adopt a Bayesian approach wherein it is assumed that the customers' private values are drawn from a known distribution. This is not always the case, and pricing heavily depends on the accuracy of the underlying market analysis. Such analysis also needs to be updated frequently in order to adapt to changes in the market. In contrast, the online Ex-CORE auction mechanism used in our pricing framework is based on a competitive auctioning framework proposed by Goldberg and Hartline [25], in which the mechanism computes a uniform price outcome when the seller knows very little about the bidders' valuations. Xu and Li [3] proposed an infinite horizon stochastic dynamic program to dynamically price the IaaS cloud provider's resources based on stochastic demand arrivals and departures of cloud users. They aim at maximizing revenue specifically for the spot market. Wang *et al*. [26] investigated the similar problem of how to set the spot price to maximize cloud provider's revenue. They present a demand curve model to capture the characteristic of spot resources and design efficient online algorithms based on the *Lyapunov optimization* technique in this regard. In order to apply their proposed pricing method, providers need to continuously monitor and analyze the demand arrival and departure rate and to devise and refine demand functions. Truong-Huu and Tham [27] formulate the competition among cloud providers as a non-cooperative stochastic game to maximize cloud providers' revenue. They provide dynamic resource pricing in which providers propose optimal price policies with regard to the current policies of other competitors. Zhang *et al.* [5] present a truthful online cloud auction mechanism building on their proposed bidding language to price and allocated resources to heterogeneous demand. The applicability of combinatorial auction mechanisms for allocation and pricing of VM instances have been also investigated by many researchers [11, 28–30]. However, combinatorial auctions have been rarely used in practice because of their complexity and difficulty of solving these problems.

Amazon Web Services has adopted an auction-like approach to expand its pricing plans with spot instances for the Amazon Elastic Compute Cloud (EC2). In this scheme, consumers communicate their bids for a VM instance hour to AWS. Subsequently, AWS reports a market-wide spot price at which VM instance use is charged, while terminating any instances that are executing under a bid price that is lower than the market price. Attempts for creating auction mechanisms have also been reported by other companies. Stokely *et al.* [31] from Google present a practical auction-based solution to the resource provisioning problem in a set of heterogeneous resource clusters. The auction determines uniform, fair resource prices that balance supply and demand. Similarly, we present a prototype of an open-source software system for selling VM machine resources in an auction-like manner. We focus on technical and architectural design aspects here, while detailed analysis of the auction mechanism exploited by the proposed framework has been extensively discussed in [9].

There are also studies investigating strategies for customers to (cost-)effectively utilize Amazon spot instances [1, 32–36]. These studies focus on running applications on spot instances, and they provide scheduling and bidding strategies to minimize cost for cloud customers, whereas this work focuses on the design and implementation of a spot market to the benefit of cloud providers and provides pricing framework to maximize the provider's revenue. These studies can utilize our proposed framework to test their techniques against dynamic pricing methods other than AWS spot instance pricing. Ben-Yehuda *et al.* [4] examined the price history of the EC2 spot market through a reverse engineering process and found that the mechanism was not completely driven by demand and supply. Their analysis suggests that spot prices are usually drawn from a tight, fixed price interval, and reflect a random non-disclosed reserve price. Javadi *et al.* [37] have provided a statistical modeling of spot prices by studying Amazon spot price traces. They performed a comprehensive analysis of spot instances in four data centers of Amazons EC2 and they proposed a statistical model that fits well in terms of spot price fluctuation and time between price changes. It seems necessary to mention that the spot instance pricing mechanism used by Amazon underwent couple of changes after these publications.

3. SYSTEM DESIGN AND IMPLEMENTATION

The aim of SipaaS is to provide an extensible framework for dynamic pricing of VM instances in a spot market based on a set of Representational state transfer (REST)ful services. By extensibility, we mean the ability to implement new pricing mechanisms and apply them in the framework without the necessity to modify the design of the web services. Different implementations of pricing mechanisms can be plugged into the framework by replacing the pricing module, which will be discussed in the later part of this paper. The implementation of the SipaaS framework encompasses the proposed auction mechanism in [9] that can be easily replaced by any other dynamic pricing mechanism. To this end, considering the scope and requirements of our proposed software system, we decided to use RESTful services for the SipaaS framework [38].

Spot instance pricing as a Service provides services for adding, removing, or updating bidders' orders (bid price plus quantity) for various types of VMs for each provider (or data center) and dynamically computes prices for each type. The SipaaS framework considers each type of VM for each cloud provider as a distinct spot market and computes prices in each market based on the submitted orders for the corresponding type. The framework is agnostic to the cloud platform and resource management system used by the cloud provider, and cloud providers are supposed to implement their own extension to make use of services provided by SipaaS.

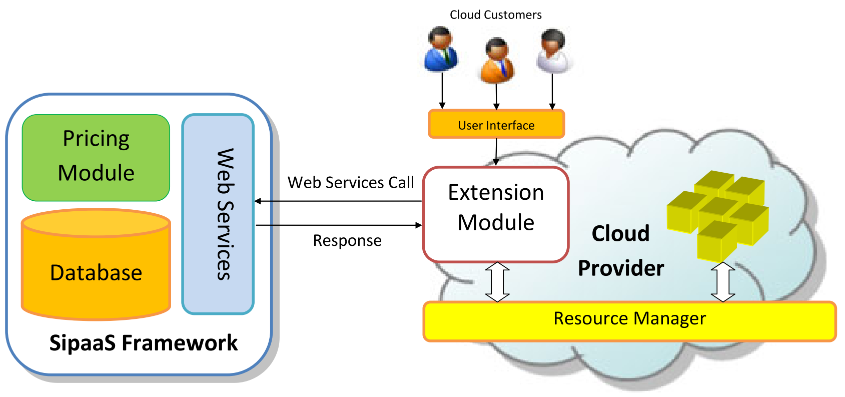


Figure 1. System model.

As shown in Figure 1, the system is composed of two main parts:

Spot instance pricing as a Service Framework: a set of RESTful services written in Java and deployed on a host to provide web services required for running the spot market. Detailed information about the web services that SipaaS offers are presented in Section 3.1.

Cloud Provider's Platform Extensions: an add-on software that must be installed as a module on the cloud provider's platform to make use of web services provided by SipaaS. Detailed information on such an extension to OpenStack is presented in Section 3.2.

As shown in Figure 1, add-on software on the provider's platform calls web services on SipaaS framework using the REST application programming interfaces (i.e., HTTP requests) and receives responses in JavaScript Object Notation (JSON) [39] format in case of successful calls or error messages in case of errors.

3.1. *Spot instance pricing as a Service framework*

SipaaS stands for Spot instance pricing as a Service. As its name implies, the main goal of SipaaS is to provide pricing for spot instances in the form of services. Thus, it has been designed based on a set of web services, by invoking them, the cloud provider is able to price the spot instances. SipaaS web services are implemented based on Spring Model–view–controller (MVC) framework [40]. As shown in Figure 2, SipaaS contains three main components:

(1) *Pricing Module*: This component is the heart of the system and embodies the technique used for pricing spot instances. The pricing module computes the market-wide single price based on the submitted orders by customers. The pricing module receives a list of orders with the *reserve price* and the *available capacity* in number of VMs, which have been set by the provider and computes the spot market price accordingly. Details of auction mechanism employed in SipaaS are given in Section 3.3.

(2) *Database*: A relational database is utilized by SipaaS to store information related to each spot market. Considering the fact that we look for a database management system (DBMS) with high-speed and reliability [41], we have chosen MySQL as a DBMS, which can be replaced by any other type of DBMS according to the requirements. The database server can be deployed either on the same host where SipaaS is installed or on a dedicated host. Figure 3 depicts the *enhanced entity relationship diagram* (EERD) of the SipaaS' database, which contains eight main tables:

    (a) provider: The *provider* table stores information about providers (or data centers), which register themselves in SipaaS. Each provider has a unique *ID*, *accesskey*, and *secretkey*. The provider might have any arbitrary *name* as well. Providers require their *accesskey* and *secretkey* to invoke web services provided by SipaaS.
    (b) vmtype: Providers might have different types of VMs for each of which a distinct spot market executes. The *VM Type* table stores information about these types for every

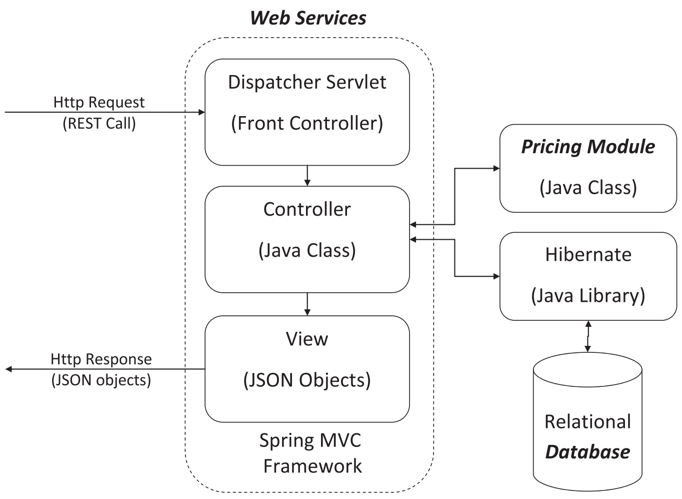


Figure 2. Spot instance pricing as a Service framework components.

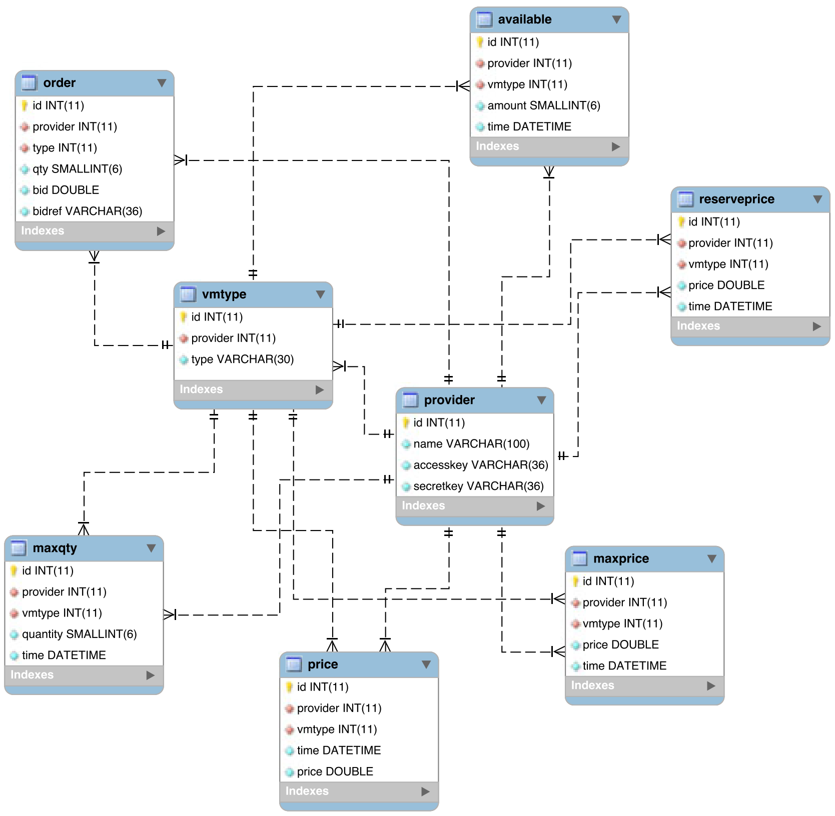


Figure 3. Enhanced entity relationship diagram of the database.

provider. The information contains: a unique ID (*id*), provider's id (*provider*), and the type name (*type*).

(c) order: The *order* table stores information about orders by customers for each VM type and each provider. The information contains: a unique ID (*id*), provider's ID (*provider*), VM type ID (*type*), the requested number of VMs (qty), bid price (*bid*), and a unique reference ID (bidref), which must be generated by the cloud provider and is used for any future reference to this order.

(d) price: The *price* table stores information about spot market price generated by the pricing module. Each price contains id, *provider*, vmtype, *time*, and *price*. The *time* attribute records the timestamp for a given date and time of day the price is computed.
(e) available: The *available* table stores data on available capacity of the provider for each spot market. The table contains: *id*, *provider*, vmtype, *amount*, and *time*. The *amount* and *time* attributes are used to store the total number of available VMs for the corresponding VM type and timestamp of setting the available capacity, respectively. If the available capacity is not set by the provider, SipaaS assumes infinite availability.

    It is worth noting that, as demand for each type of VMs can fluctuate over time, providers are supposed to dynamically adjust the capacity allocated to each spot market to match the demand in order to maximize total revenue. In the current implementation of the SipaaS framework, the provider is responsible for adjusting the spot market capacity and continuously updating the availability if demand surpasses supply. One possible future extension can be adding components to handle the dynamic capacity control for each spot market. Work similar to that of Zhang *et al*. [8] would be useful in this regard.
(f) reserveprice: The *reserveprice* table, similar to *available* table, stores data on the reserve price for each spot market. Reserve price is the lowest bid price that the provider accepts for the VM instance time slot usage (e.g., instance-hour). This table contains: *id*, *provider*, vmtype, *price*, and *time*. If the reserve price is not set by the provider, the minimum value of zero is assumed.
(g) maxprice: The *maxprice* table stores the maximum bid price acceptable by the pricing module. This table contains: *id*, *provider*, vmtype, *price*, and *time*. No maximum bid price suggests no limit on the bid price.
(h) maxqty: The maxqty table stores maximum number of VMs that can be requested by a customer (i.e., an order). This table contains: *id*, *provider*, vmtype, *quantity*, and *time*.

(3) *Web Services*: SipaaS provides a set of web services that facilitate the execution of spot markets by cloud providers. Table I shows the list of main web services available in SipaaS. All services are RESTful web services designed to produce responses in the JSON format. SipaaS utilizes RESTful web services as they are easy to implement and require minimal middleware, that is, only HTTP support is required. JSON is also a highly portable data transfer format that can be easily recognized by client applications. The cloud provider aiming at utilizing SipaaS framework services requires a clear understanding of the context as well as the content that must be passed through each web service invocation. The following parameters are mostly common among different web services:

    (a) accesskey: This is an alphanumeric text string that is uniquely assigned to the provider and identifies its owner. This parameter is used to differentiate cloud providers from each other.

Table I. Spot instance pricing as a Service framework web services.

Web service name	Input parameter(s)	Output
`Register`	`name`	`credentials`
`Unregister`	`accesskey,secretkey`	`status`
`Regvmtype`	`accesskey,secretkey,type`	`status`
`Unregvmtype`	`accesskey,secretkey,type`	`status`
`Setavailables`	`accesskey,secretkey,vmtype,quantity`	`price`
`Setmaxqty`	`accesskey,secretkey,vmtype,quantity`	`status`
`Setreserveprice`	`accesskey,secretkey,vmtype,value`	`price`
`Setmaxprice`	`accesskey,secretkey,vmtype,value`	`status`
`Addorder`	`accesskey,secretkey,vmtype,quantity,bid,ref`	`price`
`Updateorder`	`accesskey,secretkey,quantity,ref`	`price`
`Removeorder`	`accesskey,secretkey,ref`	`price`
`Pricehistory`	`accesskey,secretkey,vmtype,fromdate,todate`	`price(s)`

(b) secretkey: It plays the role of a password for the provider. A *secretkey* with *accesskey* form a secure information set that confirms the provider's identity.
(c) vmtype: It determines the type of VM or equally a specific spot market. The vmtype is a string containing the VM type name and is used to relate each request to the corresponding spot market.

The main web services provided by the framework are:

(a) *Register*: This service allows the provider to register itself with framework. It receives one string parameter called *name* as an input, representing the provider's name. In response to successful registration, the web service generates as output a pair of *accesskey* and *secrectkey* in JSON format.
(b) *Unregister*: The provider is able to unregister from SipaaS by invoking this web service.
(c) *Regvmtype*: Using this web service, the cloud provider is able to register different types of VMs in the system. In addition to *accesskey* and *secretkey*, another input parameter called *type* must be provided. The *type* parameter is a string value representing the VM type name. As stated earlier, each VM type together with a provider stands for a distinct spot market.
(d) *Unregvmtype*: As its name implies, it removes a VM type.
(e) *Setavailables*: This web service receives vmtype and *quantity* as inputs to specify the maximum available capacity in number of VMs for the specific spot market. The *Setavailables* web service can be called any time and multiple times throughout the spot market lifetime. The output of this web service is a spot market *price* computed according to the updated capacity.
(f) *Setmaxqty*: It performs similarly to *Setavailables*, whereas it specifies the maximum number of VMs user can request in one order.
(g) *Setreserveprice*: This web service performs similar to *Setavailables* and specifies the reserve price. It is important to mention that invoking *Setreserveprice* and *Setavailables* might not result in a new spot market price, in that case, the JSON response includes the same spot price as the latest one. This web service provides an option for the cloud provider to take its *costs* for delivering a VM instance into account. Using *Setre-Serveprice* service, the cloud provider can dynamically set the lowest price accepted for one slot usage of a VM instance during that time. The pricing module must ignore orders with bid price below the current reserve price.
(h) *Setmaxprice*: It specifies the maximum bid price allowed in an order. By using *Setmax-price* and *Setreserveprice*, a provider is able to limit the range of bid prices submitted by spot market users.
(i) *Addorder*: The *Addorder* web service allows providers to insert a new order (*bid price* plus *quantity*) to the system. The *ref* parameter is a unique value provided for each order and is used for future references to this order. The output of the service is the spot market *price*.
(j) *Updateorder*: The *Updateorder* web service allows providers to update a previously submitted order to the system. This web service is called when a customer terminates part of requested running instances under the specific order.
(k) *Removeorder*: The *Removeorder* web service allows provider to remove a previously submitted order. This web service is called whenever all VM instances of the accepted order are terminated.
(l) *Getpricehistory*: This web service receives the input parameters *fromdate* and *todate*, and in response provides the pricing history of a certain spot market.

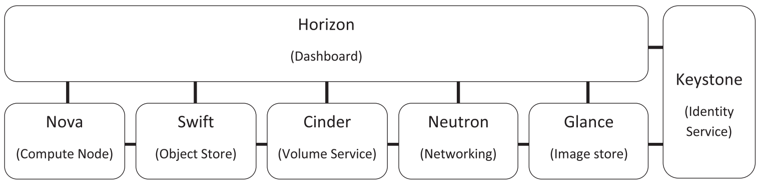


Figure 4. OpenStack components.

3.2. Extensions for Horizon – the OpenStack dashboard

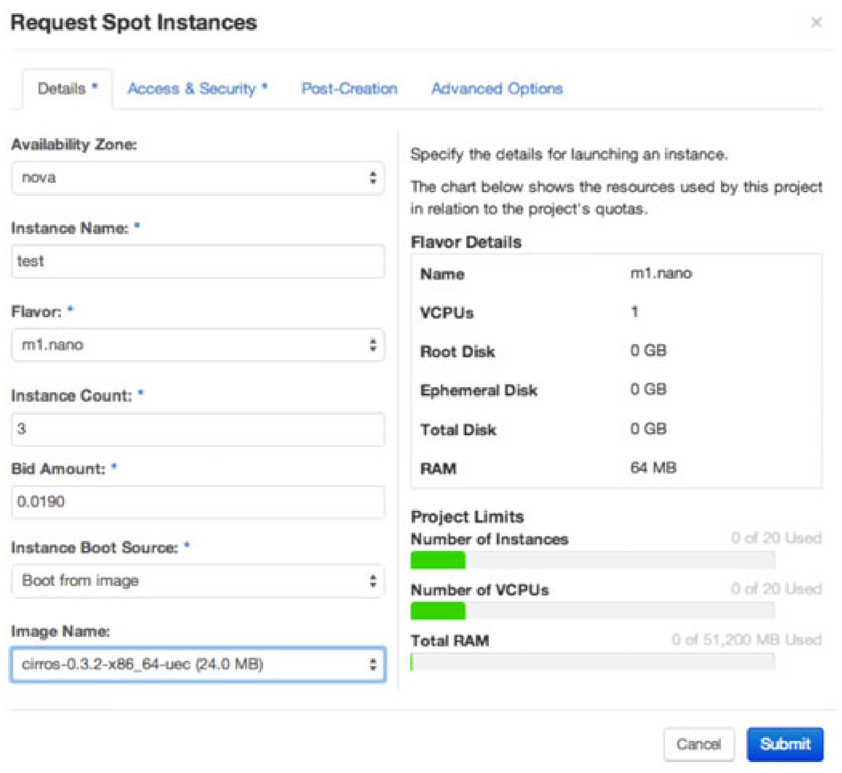
Cloud providers utilizing the SipaaS framework need to setup their own customized extension software capable of interacting with SipaaS web services. In this section, we discuss about how such an extension is designed and implemented for the *OpenStack* platform.

OpenStack is an open-source cloud management platform, developed to control pools of compute, storage, and networking resources in a data center. OpenStack has been designed as a series of loosely coupled components that are easy to integrate with a variety of solutions and hardware platforms. One of these components is *Horizon*, which provides users and administrators with management capabilities via a web interface. As schematically shown in Figure 4, Horizon provides a dashboard interface for accessing OpenStack services provided through its main components. The main components of OpenStack platform are briefly described below:

- *OpenStack Dashboard (Horizon)*: It provides a web based user interface to other services such as Nova, Swift, and Keystone. Management actions enabled by this component include VM image management, VM instance life cycle management, and storage management;
- *OpenStack Compute (Nova)*: It manages the VM instance life cycle from scheduling and resource provisioning to live migration;
- *OpenStack Storage (Swift)*: Swift is a scalable redundant storage system responsible for enabling data replication and ensuring integrity;
- *Block Storage (Cinder)*: The block storage system allows users to create block-level storage devices that can be attached to or detached from VM instances;
- *OpenStack Networking (Neutron)*: Neutron is a system for managing networks and IP addresses. The system allows users to create their own networks and assign IP addresses to VM instances;
- *OpenStack Identity (Keystone)*: Keystone is an account management service that acts as an authentication and access control system;
- *OpenStack Image (Glance)*: It supplies a range of VM image management capabilities from discovery and registration to delivery of services for disk and server images.

To add spot market facilities to OpenStack, we extended Horizon to make it capable of using the services provided by SipaaS. Horizon's main panel includes two different sections, one for members with system administration role and another section for other standard users. Because the admin section is only visible to users with administrator privileges, we added a new panel to this section through which system administrators are capable of enabling spot market support and can define global settings such as maximum and minimum amount of bid price for users, number of available VMs for allocation, and the maximum number of VMs a user can request. These parameters are passed to SipaaS by calling corresponding services defined in SipaaS and discussed in earlier parts of this section.

We also added another panel to the section visible to standard OpenStack users, labeled as *Spot Instances*. As it can be seen in Figure 5, this panel allows users to submit their bids and the number of required spot instances (i.e., orders) to the system. At the backend, when a user submits his request for using spot instances, a unique reference number is created for each order and this reference number with corresponding bid price and number of requested instances are sent to Sipaas by calling its *Addorder* service and are stored in a database table named *order*. According to the computed and returned price by SipaaS, if user's bid amount is greater than the price, which means the request can be fulfilled, requested instances are created and two local database tables named *order* and *instance*


Request Spot Instances

Details * | Access & Security * | Post-Creation | Advanced Options

Availability Zone: nova

Instance Name: * test

Flavor: * m1.nano

Instance Count: * 3

Bid Amount: * 0.0190

Instance Boot Source: * Boot from image

Image Name: cirros-0.3.2-x86_64-uec (24.0 MB)

Specify the details for launching an instance.

The chart below shows the resources used by this project in relation to the project's quotas.

Flavor Details

Name	m1.nano
VCPUs	1
Root Disk	0 GB
Ephemeral Disk	0 GB
Total Disk	0 GB
RAM	64 MB

Project Limits

Number of Instances 0 of 20 Used

Number of VCPUs 0 of 20 Used

Total RAM 0 of 51,200 MB Used

Figure 5. Screenshot of requesting spot instances web page.

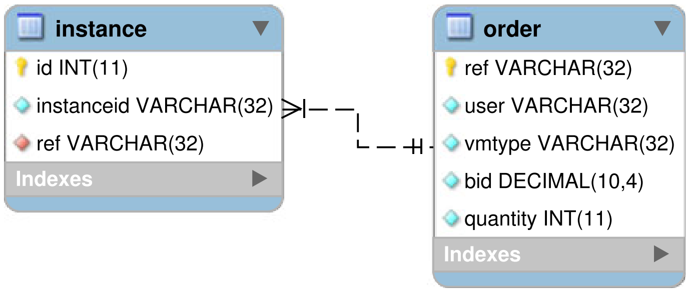


Figure 6. Enhanced entity relationship diagram of the database used for Horizon extensions.

are updated. Figure 6 shows an enhanced entity relationship diagram of the local database including *order* and *instance* tables created in the OpenStack platform.

In the *instance* table, order reference created at the previous step and instance IDs created after launching VMs are stored. Later, if a user terminates any of the spot instances, the table is updated accordingly, and the *Updateorder* service of SipaaS is invoked to calculate the new price. If a user terminates all the instances belonging to a single order, then *Removeorder* is called and after both mentioned operations, the instances of other users, which belong to an order with bid price lower than the current market price, are terminated automatically (out-of-bid orders termination). Figure 7 shows the sequence diagram of the process of handling an order submission by a user.

There is another added panel labeled *Spot Pricing History*, in which users are able to view the history of spot price fluctuations. By supplying the desired duration, users can see the plotted result from the invoked *Pricehistory* service of Sipaas. A sample screenshot of the spot pricing history panel is depicted in Figure 8.

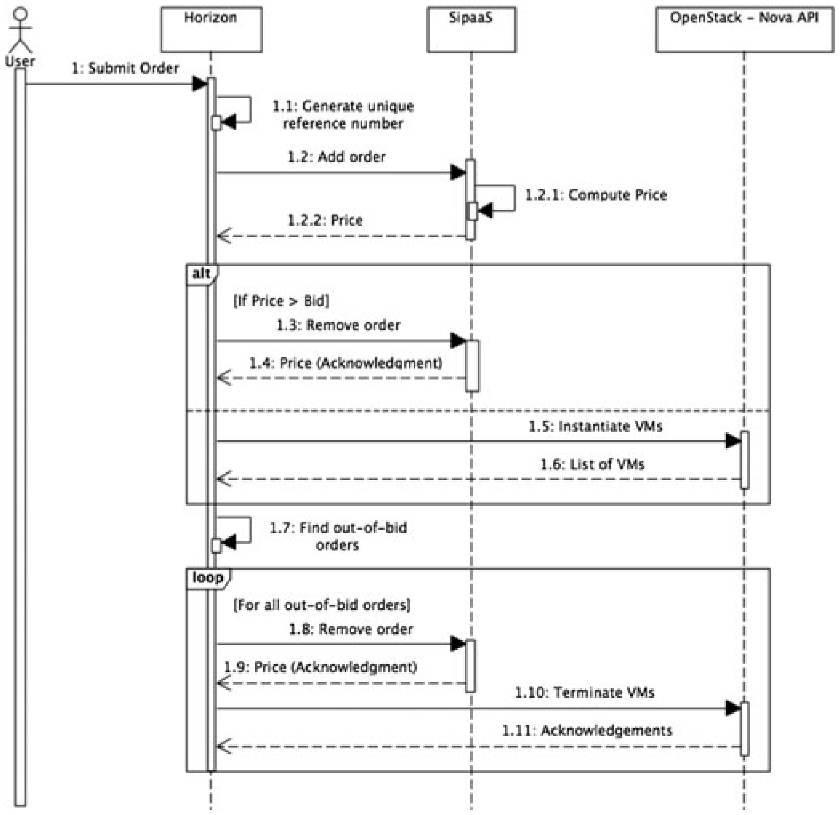


Figure 7. Sequence diagram of an order submission handling.

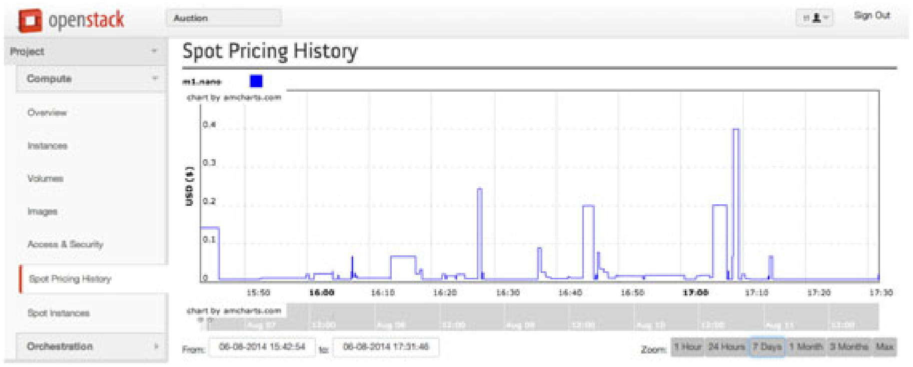


Figure 8. Screenshot of spot pricing history web page.

3.3. Pricing mechanism

This section discusses the implemented algorithm in the pricing module of SipaaS. The pricing mechanism plugged into the SipaaS framework works based on the online Ex-CORE auction algorithm proposed in [9]. Here, we discuss the general concepts of online Ex-CORE auction while details of the proposed auction can be found in [9].

The Ex-CORE algorithm generates a market-wide single price for each auction round according to the current available orders in the market. The main aim of the Ex-CORE algorithm is to maximize the provider's revenue while it is strategy-proof (truthful). An auction mechanism is *strategy-proof* – also called truthful or incentive-compatible – if the *dominant* bidding strategy for every bidder is to always report their true valuation irrespective of the behavior of the other bidders. It is shown that the Ex-CORE auction has a high probability of being truthful, and generates a near optimal profit for the provider in each round of auction [9].

To maximize revenue, the random component of the auction mechanism must generate the price in a way that the gained revenue is a good estimate of the revenue generated by the *optimal single-price* auction. The *optimal single-price* auction, $\mathcal{F}$, is defined as follows:

Definition 1
Let $\mathbf{d}$ be an order vector. An order, d_i, is a pair of (r_i, b_i), where r_i represents the number of required VMs and b_i the bid price. Without loss of generality, suppose the components of $\mathbf{d}$ are sorted in descending order by bid prices. We denote by $\sigma_k(\mathbf{d})$ the sum of the number of requested VM instances in the sorted vector of orders from the first order to kth order $\left(\sigma_k(\mathbf{d}) = \sum_{i=1}^{k} r_i\right)$. The auction $\mathcal{F}$ on input $\mathbf{d}$ determines the value k such that $b_k\sigma_k(\mathbf{d})$ is maximized. All bidders with $b_i \geqslant b_k$ win at price b_k and all remaining bidders lose. Thus, the revenue of $\mathcal{F}$ on input $\mathbf{d}$ is

$$\mathcal{F}(\mathbf{d}) = \max_i \ b_i\sigma_i(\mathbf{d})\,. \tag{1}$$

The *optimal single-price* auction sets the sale price for a set of orders as follows:

Definition 2
Denote $opt(\mathbf{d})$ the optimal single sale price for $\mathbf{d}$ that maximizes the revenue for the auctioneer, that is,

$$opt(\mathbf{d}) = \text{argmax}_{b_i} \ b_i\sigma_i(\mathbf{d})\,. \tag{2}$$

All bidders with $b_i \geqslant opt(\mathbf{d})$ win the auction and pay $opt(\mathbf{d})$ and all remaining bidders lose and pay 0. Even though, the *optimal single-price* maximizes revenue for the provider, it is not truthful and allows for the increase in the cloud user's utility by misreporting the true valuation of the service. Therefore, we propose Ex-CORE auction, which generates a near optimal profit for the provider while is truthful with high probability.

Algorithm 1 outlines the online version of Ex-CORE that attempts to maximize the revenue in a greedy manner given a newly arriving order and the existing orders. The online Ex-CORE records the optimal sale price computed by $opt(\mathbf{d})$ in the previous round of algorithm and updates the sale price using the Ex-CORE algorithm only when the optimal sale price calculated in the current round differs from the one in the previous round of the auction. Lines 1–4 of the algorithm enforce this

Algorithm 1 The Online Ex-CORE Auction

```
Input: $\mathbf{d}, p_{cur}, p_{optprv}$   ▷ $\mathbf{d}$ is the list of orders, sorted in descending order of bids, $p_{cur}$ is current
    market price, $p_{optprv}$ is the optimal single price in the previous round.
Output: $p$                                                        ▷ Sale Price
 1: $p_{opt} \leftarrow opt(\mathbf{d})$
 2: if $p_{opt} = p_{optprv}$ then
 3:     return $p_{cur}$
 4: end if
 5: $r \leftarrow$ the largest $r_i$ in $\mathbf{d}$
 6: $m \leftarrow \underset{\sigma_i(\mathbf{d})}{\text{argmax}} \ b_i\sigma_i(\mathbf{d})$
 7: if $m \leq r$ then
 8:     return $p_{opt}$                                           ▷ single optimal price
 9: else
10:     $\rho \leftarrow \frac{m}{m-r}$
11:     Find $c$ in $\rho\, ln(c) + \rho - c = 0$
12:     $u \leftarrow rnd(0,1)$                          ▷ chosen uniformly random on [0,1]
13:     $l \leftarrow \lfloor log_c(\mathcal{F}(\mathbf{d})) - u \rfloor$
14:     $R \leftarrow c^{(l+u)}$
15:     $j \leftarrow$ the largest $k$ such that $\frac{R}{\sigma_k(\mathbf{d})} \geq b_k$
16:     return $\frac{R}{\sigma_j(\mathbf{d})}$
17: end if
```

rule. Lines 5 and 6 compute r, the maximum number of requested units in the order list, and m, the maximum number of units sold by *optimal single-price*. The Ex-CORE mechanism is designed to work for mass-market scenarios that requires m to be larger than r ($m \gg r$). In the rare event when this condition would not hold, the algorithm returns the price computed by *optimal single-price*. In line 10, ρ is computed, followed by the computation of the optimal value for c in line 11. Subsequently, c is used to generate an estimation of $\mathcal{F}(.)$ (lines 12–14). Finally, the estimated value is converted to the market clearing price through the revenue extraction mechanism. We omit further details and proofs regarding the online Ex-CORE auction in this paper, and the interested readers are referred to [9] for complementary details.

To retrieve price, web services in SipaaS call the auction algorithm in the pricing module, which is the online Ex-CORE in the current study. For example, each time *Addorder*, *Updateorder*, and *Removeorder* web services are invoked, the auction algorithm is called to generate the current price based on the newly updated order vector. The online Ex-CORE records the optimal sale price computed by *optimal single-price* auction in each round and updates the sale price only when the optimal sale price calculated in the current round differs from the one in the previous round of the auction, that is, pricing module does not update the sale price by receiving every order requests. This prevents the market from being exposed to high price fluctuations because of a random component in the Ex-CORE algorithm, which calculates the sale price [9].

4. PERFORMANCE EVALUATION

Our evaluation of the proposed framework is twofold. In the first, an experimental study is conducted to validate and test the framework. We also analyze behavior of participants in the study. The second part evaluates the scalability of the framework under high service loads. Even though our experiments inevitably leads to performance evaluation of the Ex-CORE auction mechanism as the heart of the system, our aim here is to show that our proposed architecture can operate in real environment settings. Detailed performance evaluation of the utilized auction mechanism can be found in [9].

4.1. Validation and truthful analysis

In this section, we evaluate our proposed framework by conducting an experimental study in a real environment. The goals are twofold: (i) to demonstrate that the extension to the OpenStack combined with the SipaaS framework can operate in a practical environment to provide spot market facilities and (ii) to evaluate the system's behavior and our auction pricing model in a test experiment.

4.1.1. Experimental testbed. The testbed used for the evaluation of the system consists of two main hosts, both running Ubuntu 14.04 as operating system. One was used for running SipaaS and the other one was used for running all the OpenStack services. The *DevStack OpenStack*,[‖] which is suitable for development and operational testing, is used in the experiment. Hosts used in the experiment are `m1.small` and `m3.2xlarge` VM instances running on Amazon EC2 in `Asia Pacific - Sydney` region. Resource configuration of VM instance types used in the experiment can be seen in Table II.

The `m1.small` machine was chosen to deploy SipaaS web services on Apache Tomcat 6 web server[**] and MySQL as a DBMS to host and manage the database. The `m3.2xlarge` machine was used to host all the OpenStack services including Horizon and its extension, deployed on Apache HTTP server version 2. MySQL was also installed in this machine to host the database for storing all the required data for the extension software.

In the experimental study, users can use their own desktop or laptop PC with a web browser (preferably Google Chrome) to connect to the dashboard and use the spot market services.

[‖]Deploying OpenStack for Developers, http://devstack.org/.
[**]Apache Tomcat, http://tomcat.apache.org/.

Table II. Types of virtual machine instances and their specifications used to host system components in the experiment.

Instance type	Cores	CPU (ECU*)	Memory (GB)	Storage (GB)
`m1.small`	1	1	1.7	160
`m3.2xlarge`	8	26	30	160

One ECU (EC2 compute unit) is equivalent to CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor.

Table III. True private values of experiment participants.

User	Price value ($)	Quantity
T1, C1	0.0691	2
T2, C2	0.0092	1
T3, C3	0.0475	1
T4, C4	0.0232	2
T5, C5	0.0184	1

4.1.2. Experimental design and setup. We conducted a 20-min experiment with 10 participants (i.e., spot market users). Participants were divided into two groups of five: (i) Group T or truthful bidders and (ii) Group C or counterpart bidders who have the freedom to misreport their true private values to maximize their utility. Participants of the latter are selected from a group of professional cloud users who have satisfactory level of knowledge about the spot market. Each participant was provided with a user-name and password to access the OpenStack dashboard. We provided participants of the experiment with a pair of uniformly random generated quantity and price values that must be considered as their true private values.

Considering the scale of the experiment, we limited the maximum number of simultaneous VM instances each user can run in the system to 2. Accordingly, we chose uniformly random true quantity values from $\{1, 2\}$. For price values, we adopted the pricing details of Amazon EC2 `m3.medium` in the `Asia Pacific-Sydney` region at the time of the experiment, while we replaced per hour charging period with 30 s in our experiment. True price values in dollars are drawn from a uniform distribution of the interval [0.0081, 0.0700], where \$0.0081 and \$0.0700 are the minimum spot instance (reserve price) and the on-demand price per hour for `m3.medium` instances in Amazon, respectively. The maximum possible bid price is also derived from the maximum allowed bid price in Amazon EC2 for the same type of spot instances, that is, \$0.4520/h.

Assuming that the provider offers on-demand pricing concurrently and Quality of Service (QoS) for spot instances are lower than equivalent on-demand ones, true values higher than on-demand price seems unreasonable. Therefore, we distributed true private price values between the minimum spot price and on-demand price for `m3.medium` instances. However, experiment participants are allowed to submit orders with a bid price higher than the on-demand price. Table III shows true private price and quantity values for participants of each group.

Similar to Amazon, spot instances are not charged for their partial 30 s upon their termination by the provider, while a partial 30 s of usage is counted as a full 30 s upon termination by the user. Each full time slot usage (i.e., 30 s) is charged based on the spot market price at the beginning of the time slot.

Participants of the experiment are provided with the details of the online Ex-CORE auction algorithm and their utility function for one time slot instance usage, formulated as follows:

$$u(r,b) = \begin{cases} (qv - rp)x\,, & \text{if } b \geqslant p \text{ and } r \geqslant q; \\ 0\,, & \text{otherwise.} \end{cases} \tag{3}$$

where r, b, q, v, p, and x are the requested number of instances, bid price value, true private quantity value, true private price value, spot market price at the time of order submission, and a Boolean value describing whether the order is accepted or not, respectively.

Participants are asked to acquire VM instances of type `m1.nano` as long as they can, according to their true private values using the OpenStack dashboard. Participants of group T are obliged to submit their true values to acquire instances through the whole experiment regardless of the market price fluctuation. Participants of group C are asked to try to maximize their utility based on rules of the experiment and given pricing information. Therefore, if it is deemed beneficial, a participant of group C might strategically misreport her/his bid price or the quantity, that is, $b \neq v$ or $r \neq q$. To provide enough incentives for participant of group C to act rationally in the experiment, we considered a prize for the winner of the experiment. The winner of the experiment is the one who can achieve the highest positive difference of the utility value with his/her counterpart truthful bidders.

All participants are taught the goal, rules, and details of the experiment. The experiment was conducted in the real environment where participants have been able to run instances according to their order submissions. The results of the experiment are discussed in the following section.

4.1.3. Results and analysis. The main goal of the conducted experiment is to show that the system works flawlessly in a practical test environment. All the participants experienced valid system behavior in the experiment. They were able to submit their orders to the system and boot up their instances whenever their bid was higher than the market price. As soon as market price went above of the submitted bid price, acquired instances were terminated by the system immediately without any prior notice.

Figure 9 depicts the market price fluctuation during the experiment. As shown in the figure, the price reaches the maximum bid price in multiple cases. This happened because of the reason that some low value participants, for example $C4$ and $C2$, who were starving in the market, tried to terminate other participants' instances by submitting very high bid price and terminating instances of others immediately to submit their new orders. This, however, affected their utility value because they were charged multiple times higher than their true values.

In [9], it is theoretically proved that Ex-CORE auction mechanism is truthful with high probability. Therefore, as we expected, excluding $T3$, all truthful users (i.e., participants of group T) achieved higher utility value than their counterpart users who misreported their true values. Table IV shows the total cost and achieved utility values by all users based on the utility function in Equation (3).

In order to investigate how user $C3$ could achieve the highest positive difference in comparison with his/her paired truthful participant, we analyzed the submitted orders by all users. The result of our analysis shows that $C3$ is the most truthful user among the participants of group C, who continuously submitted the true quantity value and bid price values significantly close to his/her true value. The only reason $C3$ achieved highest difference is that she/he was quicker in submitting orders and could obtain two additional full time slots of instance usage. The truthful user $T3$ was also able to do the same if he/she would submit his/her true values fast enough at the same time.

As it can be seen in Table IV, $T2$ and $T5$ could not acquire instances for a full time slot at all, because the market price was often higher than their true price values. $T4$ similarly ends up running instances for only one time slot. Comparatively, paired users from group C acquired instances for higher number of time slots. However, their overall utility values are negative as they ended up paying more than their true values.

Results of the experiment reported in Table IV support the theoretically proven supposition that Ex-CORE algorithm is truthful with high probability. This confirms the fact that rational users' dominant strategy in a truthful auction mechanism is to report their true private values. Moreover, our investigation on the historical price data of spot instances in Amazon EC2 shows that price spikes similar to what happened in our experiment are occurring in Amazon's spot market as well. This might happen either because of the same experience we had in our experiment where some users submit very high bids or possibly sudden spikes in demand. Intuitively, without knowing how the spot market mechanism works, no user has the incentive to strategize over its bid. This has been suggested by other studies as well [2, 4].

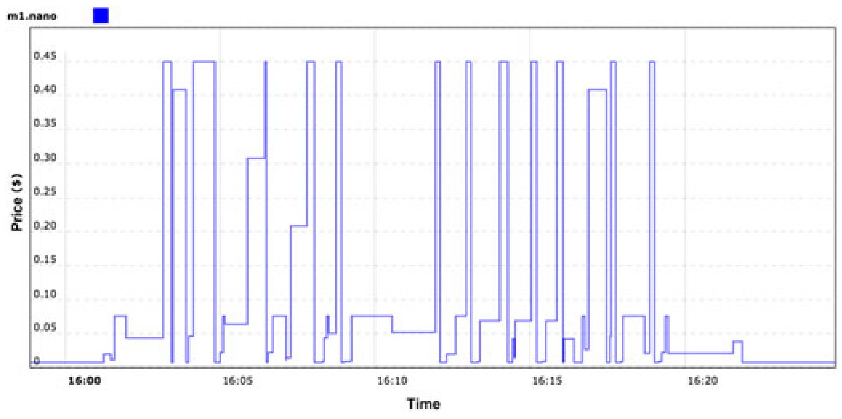


Figure 9. Spot market price fluctuation during the experiment.

Table IV. Total cost, the number of full time slots usage, and utility values of experiment participants.

User	Total cost ($)	Number of full time slots	Utility value ($)
T1	1.2964	16	**0.9148**
C1	1.8216	17	0.5278
T2	0.0000	0	**0.0000**
C2	10.0227	18	−9.8571
T3	0.1896	6	0.0954
C3	0.2280	8	**0.1520**
T4	0.0436	1	**0.0030**
C4	3.6810	5	−3.4490
T5	0.0000	0	**0.0000**
C5	0.0738	2	−0.0370

4.2. *Performance analysis*

In this section, first we discuss the scalability of the framework, and then we conduct an experiment to evaluate it. As we mentioned earlier, SipaaS provides a set of web services for cloud providers to run spot markets. Similar to other web applications, SipaaS performance depends on the performance of the hosting web server and used database management system on which the framework is installed; they are Apache Tomcat and MySQL, respectively. SipaaS follows the three-tier architecture and can use load balancers, multiple application servers, and multiple databases. Our aim here is not to represent and discuss the design and architecture of such systems. Interested readers are referred to works addressing this topic [42–44].

Response time is the main performance metric taken into account in the design and implementation of service-oriented systems such as SipaaS. Thus, we evaluate the scalability of the system in terms of response when demand (i.e., number of orders) grows. The pricing module of the SipaaS framework that embodies the technique used for dynamic pricing is the most compute-intensive part of the system. Therefore, the computation taking place in the pricing module is considered as the main factor for determination of the response time, apart from other factors such as network delay and delay between the web server and database server. We conduct an experiment to evaluate how the system performs in terms of response time when number of orders increases.

4.3. *Experimental setup and evaluation*

To evaluate the responsiveness of the framework, we developed a *web robot* (bot) application that generates order requests and submits them to the framework. It generates RESTful requests based on application programming interfaces of SipaaS and measures the response time delay for each request. To generate order requests, we adopt the following probability distributions:

– Bid prices: bid prices are drawn from a uniform distribution bounded by $(0, \$0.4500)$. The upper bound value is derived from the maximum allowed bid price for `m3.medium` in Amazon EC2 at the time of writing the paper.
– Quantity: the value for the requested number of instances in each order is drawn from a uniform distribution bounded by $[1, 20]$, where 20 is the same limit applied to spot instances by Amazon EC2.
– Arrival time: the arrival time of the order requests is generated independently following a Poisson process with parameter λ set at the total number of requests divided by the number of hour in the simulation.
– Holding time: when a user acquires instances, the time a VM instance remains active in the system (holding time) if the bid is valid is modeled by Pareto distributed random variables, with shape parameter 1 and location parameter 1.

The testbed used to deploy SipaaS web services for performance evaluation consisted of an HP EliteDesk 800 machine with following the hardware specifications:

– Intel(R) Core(TM) i7 Processors @ 3.6 GHz.
– 16 GB, 1600 MHz DDR3 SDRAM.
– Seagate ST500LM000 HPDA WU 500 GB.

All required application components, including Tomcat7, MySQL, and the Bot software, were installed on this machine to minimize the effect of factors such as network delay. The number of order requests submitted by bot software is increased from 10 to 100,000 by multiplies of 10 for a period of 2 h. The response time is measured by measuring the time frame between an order request submission and the reception of the market price as a response. Figure 10 plots the impact of number of submitted orders on the response time of SipaaS framework on the testbed.

As it can be seen, for 1000 requests per 2 h and below, the response time remains almost constant at around 10 ms. However, by increasing the number of requests above 1000, the response time of the system increases until on average it reaches 62 ms at 100 thousands requests, which is negligible compared with the provisioning time of a VM instance. The experiment shows that the pricing module scales reasonably when demand grows. One needs to be careful about the computational complexity of pricing algorithms plugged into the framework. Otherwise, the simple architecture of the SipaaS framework imposes negligible overhead to the responsiveness of the system.

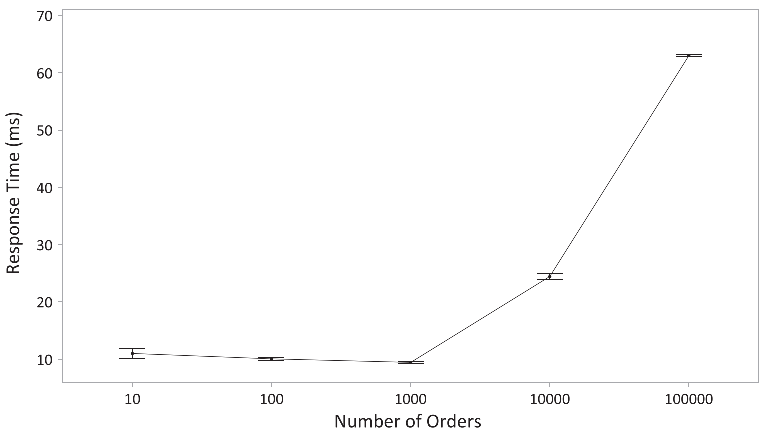


Figure 10. The mean and 95% confidence interval of the response time by the SipaaS framework against the number of submitted orders. Individual standard deviations are used to calculate the 95% confidence interval intervals. The x-axis is in log scale.

5. CONCLUSIONS

In this paper, we introduced an open source SipaaS framework that provides a set of web services to facilitate running a spot market. We also presented our extension to Horizon – OpenStack dashboard project – that makes use of the SipaaS framework to run a spot market in the OpenStack platform. In order to evaluate and validate our proposed system, we conducted an experimental study with a group of 10 participants. The results of the experimental study support the validity of the proposed system and demonstrates the behavior of the system. In addition, our study experimentally confirms the truthfulness of the auction pricing mechanism used in the SipaaS framework. As a summary, those IaaS cloud providers interested to run spot market resembling the Amazon EC2 spot instances could consider our proposed SipaaS framework using the Ex-CORE auction algorithm as a relevant replacement. The provider would expect the same user behavior using either Amazon EC2 spot instance pricing or our method, as users do not know how the spot price reacts according to their orders in both cases.

6. SOFTWARE AVAILABILITY

Spot instance pricing as a Service is an open source project, and its source is available at Bitbucket (https://bitbucket.org/farzadkh/sipaas).

ACKNOWLEDGEMENTS

This work was partially supported by a grant from the Australian Research Council (ARC). The authors would like to thank the CLOUDS lab members in particular Safiollah Heidari, Atefeh Khosravi, Nikolay Grozev, Yaser Mansouri, Le Tan Chanh Nguyen, Deepak Poola, and Yali Zhao for their participation in the experimental study.

REFERENCES

1. Song Y, Zafer M, Lee K-W. Optimal bidding in spot instance market. *2012 Proceedings IEEE Infocom*, 2012; 190–198.
2. Wang W, Li B, Liang B. Towards optimal capacity segmentation with hybrid cloud pricing. *Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems (ICDCS'12)*, Macau, China, 2012; 425–434.
3. Xu H, Li B. Dynamic cloud pricing for revenue maximization. *IEEE Transactions on Cloud Computing* 2013; **1**(2):158–171.
4. Ben-Yehuda OA, Ben-Yehuda M, Schuster A, Tsafrir D. Deconstructing Amazon EC2 spot instance pricing. *ACM Transaction Economy Computing* 2013; **1**(3):16:1–16:20.
5. Zhang H, Li B, Jiang H, Liu F, Vasilakos AV, Liu J. A framework for truthful online auctions in cloud computing with heterogeneous user demands. *Proceedings of Infocom* 2013; 1510–1518.
6. Wang W, Liang B, Li B. Revenue maximization with dynamic auctions in IaaS cloud markets. *Proceedings of the 21st IEEE/ACM International Symposium on Quality of Service (IWQoS)*, 2013; 1–6.
7. Son S, Sim KM. A price- and-time-slot-negotiation mechanism for cloud service reservations. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 2012; **42**(3):713–728.
8. Zhang Q, Zhu Q, Boutaba R. Dynamic resource allocation for spot markets in cloud computing environments. *Proceedings of the Fourth IEEE International Conference on Utility and Cloud Computing (UCC'11)*, Melbourne, Australia, 2011; 178–185.
9. Toosi AN, Vanmechelen K, Khodadadi F, Buyya R. An auction mechanism for cloud spot markets. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 2016; **11**(1).
10. Xu H, Li B. A study of pricing for cloud resources. *SIGMETRICS Performance Evaluation Review* 2013; **40**(4):3–12.
11. Zaman S, Grosu D. Combinatorial auction-based allocation of virtual machine instances in clouds. *Journal of Parallel and Distributed Computing* 2013; **73**(4):495–508.
12. Lampe U, Siebenhaar M, Papageorgiou A, Schuller D, Steinmetz R. Maximizing cloud provider profit from equilibrium price auctions. *Proceedings of 5th IEEE International Conference on Cloud Computing (CLOUD'12)*, Honolulu, Hawaii, USA, 2012; 83–90.
13. Feng Y, Li B, Li B. Price competition in an oligopoly market with multiple IaaS cloud providers. *IEEE Transactions on Computers* 2014; **63**(1):59–73.
14. Zheng Z, Gui Y, Wu F, Chen G. Star: strategy-proof double auctions for multi-cloud, multi-tenant bandwidth reservation. *IEEE Transactions on Computers* 2015; **64**(7):2071–2083.
15. Samaan N. A novel economic sharing model in a federation of selfish cloud providers. *IEEE Transactions on Parallel and Distributed Systems* 2014; **25**(1):12–21.

16. Mihailescu Marian, Teo YM. Dynamic resource pricing on federated clouds. *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID'10)*, Melbourne, Australia, 2010; 513 –517.
17. Toosi Adel Nadjaran. On the economics of infrastructure as a service cloud providers: pricing, markets, and profit maximization. *Ph.D. Thesis*, 2014.
18. An B, Lesser V, Irwin D, Zink M. Automated negotiation with decommitment for dynamic resource allocation in cloud computing. *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*, Toronto, Canada, 2010; 981–988.
19. Kantere V, Dash D, Francois G, Kyriakopoulou S, Ailamaki A. Optimal service pricing for a cloud cache. *IEEE Transactions on Knowledge and Data Engineering* 2011; **23**(9):1345–1358.
20. Niyato D, Chaisiri S, Lee B-S. Economic analysis of resource market in cloud computing environment. *Proceedings of IEEE Asia-Pacific Services Computing Conference (APSCC'09)*, Biopolis, Singapore, 2009; 156–162.
21. Püschel T, Neumann D. Management of cloud infrastructures: policy-based revenue optimization. *Proceedings of International Conference on Information Systems (ICIS'09)*, Phoenix, Arizona, 2009; 2303–2314.
22. Macías M, Fitó JO, Guitart J. Rule-based SLA management for revenue maximisation in cloud computing markets. *Proceedings of International Conference on Network and Service Management (CNSM'10)*, Niagara Falls, Canada, 2010; 354–357.
23. Parsons S, Rodriguez-Aguilar JA, Klein M. Auctions and bidding: a guide for computer scientists. *ACM Computing Survey* 2011; **43**(2):10:1–10:59.
24. Myerson RB. Optimal auction design. *Mathematics of Operations Research* 1981; **6**(1):58–73.
25. Goldberg AV, Hartline JD, Karlin AR, Saks M, Wright A. Competitive auctions. *Games and Economic Behavior* 2006; **55**(2):242 –269.
26. Wang P, Qi Y, Hui D, Rao L, Liu X. Present or future: optimal pricing for spot instances. *IEEE 33rd International Conference on Distributed Computing Systems (ICDCS'13)*, 2013; 410–419.
27. Truong-Huu T, Tham C-K. A novel model for competition and cooperation among cloud providers. *IEEE Transactions on Cloud Computing* 2014; **99**(PrePrints):1.
28. Shi W, Zhang L, Wu C, Li Z, Lau FCM. An online auction framework for dynamic resource provisioning in cloud computing. *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '14)*, ACM, 2014; 71–83.
29. Samimi P, Teimouri Y, Mukhtar M. A combinatorial double auction resource allocation model in cloud computing. *Information Sciences* 2014. DOI: 10.1016/j.ins.2014.02.008.
30. Srinivasan K, Fujita S. Truthful allocation of virtual machine instances with the notion of combinatorial auction. *Proceedings of the Second International Symposium on Computing and Networking (CANDAR '14)*, 2014; 586–590.
31. Stokely M, Winget J, Keyes E, Grimes C, Yolken B. Using a market economy to provision compute resources across planet-wide clusters. *Proceedings of IEEE International Symposium on Parallel Distributed Processing (IPDPS'09)*, Rome, Italy, 2009; 1–8.
32. Yi S, Kondo D, Andrzejak A. Reducing costs of Spot instances via checkpointing in the Amazon Elastic Compute Cloud. *In proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD '10)*, Washington, USA, 2010; 236–243.
33. Voorsluys W, Buyya R. Reliable provisioning of spot instances for compute-intensive applications. *Proceedings of 26th International Conference on Advanced Information Networking and Applications (AINA'12)*, Fukuoka, Japan, 2012; 542–549.
34. Chohan N, Castillo C, Spreitzer M, Steinder M, Tantawi A, Krintz C. See spot run: using spot instances for mapreduce workflows. *Proceedings of the 2nd Usenix Conference on Hot Topics in Cloud Computing*, USENIX Association, 2010.
35. Poola D, Ramamohanarao K, Buyya R. Fault-tolerant workflow scheduling using spot instances on clouds. *Procedia Computer Science, 2014 International Conference on Computational Science* 2014; **29**(0):523–533.
36. Song K, Yao Y, Golubchik L. Exploring the profit-reliability trade-off in Amazon's spot instance market: a better pricing mechanism. *21st IEEE/ACM International Symposium on Quality of Service (IWQOS'13)*, 2013; 1–10.
37. Javadi B, Thulasiram RK, Buyya R. Characterizing spot price dynamics in public cloud environments. *Future Generation Computer Systems* 2013; **29**(4):988 –999. Special Section: Utility and Cloud Computing.
38. Pautasso C, Zimmermann O, Leymann F. Restful web services vs. "big"' web services: making the right architectural decision. *Proceedings of the 17th International Conference on World Wide Web (WWW '08)*, ACM, Beijing, China, 2008; 805–814.
39. Crockford Douglas. *The Application/JSON Media Type for JavaScript Object Notation (JSON)*, 2006.
40. Johnson R, Hoeller J, Arendsen A, Thomas R. *Professional Java Development with the Spring Framework*. John Wiley & Sons: Indianapolis, IN, US, 2009.
41. Di Giacomo M. MySQL: lessons learned on a digital library. *IEEE Software* 2005; **22**(3):10–13.
42. Fowler M. *Patterns of Enterprise Application Architecture*. Addison-Wesley: New York, NY, USA, 2003.
43. Ramirez AO. Three-Tier architecture. *Linux Journal* 2000; **2000**(75).
44. Grozev N, Buyya R. Multi-cloud provisioning and load distribution for three-tier applications. *ACM Transactions on Autonomous and Adaptive Systems* 2014; **9**(3):13:1–13:21.