Journal of Network and Computer Applications 65 (2016) 167–180

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances

Chenhao Qu*, Rodrigo N. Calheiros, Rajkumar Buyya

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

ARTICLE INFO

Article history:
Received 17 September 2015
Received in revised form
2 December 2015
Accepted 1 March 2016
Available online 5 March 2016

Keywords:
Cloud computing
Auto-scaling
Web application
Fault tolerant
Cost
QoS
Spot instance

ABSTRACT

Cloud providers sell their idle capacity on markets through an auction-like mechanism to increase their return on investment. The instances sold in this way are called spot instances. In spite that spot instances are usually 90% cheaper than on-demand instances, they can be terminated by provider when their bidding prices are lower than market prices. Thus, they are largely used to provision fault-tolerant applications only. In this paper, we explore how to utilize spot instances to provision web applications, which are usually considered as availability-critical. The idea is to take advantage of differences in price among various types of spot instances to reach both high availability and significant cost saving. We first propose a fault-tolerant model for web applications provisioned by spot instances. Based on that, we devise novel cost-efficient auto-scaling polices that comply with the defined fault-tolerant semantics for hourly billed cloud markets. We implemented the proposed model and policies both on a simulation testbed for repeatable validation and Amazon EC2. The experiments on the simulation testbed and EC2 show that the proposed approach can greatly reduce resource cost and still achieve satisfactory Quality of Service (QoS) in terms of response time and availability.

1. Introduction

There are three common pricing models in current Infrastructure-as-a-service (IaaS) cloud providers, namely *on-demand*, in which acquired virtual machines (VMs) are charged periodically with fixed rates, *reservation*, where users pay an amount of up-front fee for each VM to secure availability of usage and cheaper price within a certain contract period, and the *spot*.

The spot pricing model was introduced by Amazon to sell their spare capacity in open market through an auction-like mechanism. The provider dynamically sets the market price of each VM type according to real-time demand and supply. To participate in the market, a cloud user needs to give a bid specifying number of instances for the type of VM he wants to acquire and the maximum unit price he is willing to pay. If the bidding price exceeds the current market price, the bid is fulfilled. After getting the required spot VMs, the user only pays the current market prices no matter how much he actually bids, which results in significant cost saving compared to VMs billed in on-demand prices (usually only 10% to 20% of the latter) (http://aws.amazon.com/ec2/spot-instances/). However, obtained spot VMs will be terminated by cloud provider whenever their market prices rise beyond the bidding prices.

Such model is ideal for fault-tolerant and non-time-critical applications such as scientific computing, big data analytics, and media processing applications. On the other hand, it is generally believed that availability- and time-critical applications, like web applications, are not suitable to be deployed on spot instances.

Adversely in this paper, we illustrate that, with effective fault-tolerant mechanism and carefully designed policies that comply with the fault-tolerant semantics, it is also possible to reliably scale web applications using spot instances to reach both high QoS and significant cost saving.

Spot market is similar to a stock market that, though possibly following the general trends, each listed item has its distinctive market behavior according to its own supply and demand. In this kind of market, often price differences appear with some types of instances sold in expensive prices due to high demand, while some remaining unfavored leading to attractive deals. Fig. 1 depicts a period of Amazon EC2's spot market history. Within this time frame, there were always some spot types sold in discounted prices. By exploiting the diversity in this market, cloud users can utilize spot instances as long as possible to further reduce their cost. Recently, Amazon introduced the Spot Fleet API (https://aws.amazon.com/blogs/aws/ new-resource-oriented-bidding-for-ec2-spot-instances/), which allows users to bid for a pool of resources at once. The provision of resources is automatically managed by Amazon using combination of spot instances with lowest cost. However, it still lacks fault-tolerant capability to avoid availability

* Corresponding author.

http://dx.doi.org/10.1016/j.jnca.2016.03.001
1084-8045/© 2016 Elsevier Ltd. All rights reserved.

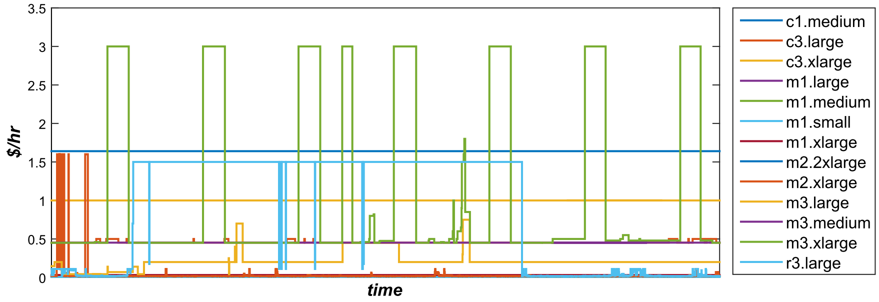


Fig. 1. One week spot price history from March 2nd 2015 18:00:00 GMT in Amazon EC2's ***us−east−1d*** Availability Zone.

and performance impact caused by sudden termination of spot instances, and thus, is not suitable to provision web applications.

To fill in this gap, we aim to build a solution to cater this need. We proposed a reliable auto-scaling system for web applications using heterogeneous spot instances along with on-demand instances. Our approach not only greatly reduces financial cost of using cloud resources, but also ensures high availability and low response time, even when some types of spot VMs are terminated unexpectedly by cloud provider simultaneously or consecutively within a short period of time.

The *main contributions* of this paper are

- a fault-tolerant model for web applications provisioned by spot instances;
- cost-efficient auto-scaling policies that comply with the defined fault-tolerant semantics using heterogeneous spot instances;
- event-driven prototype implementations of the proposed auto-scaling system on CloudSim (Calheiros et al., 2011) and Amazon EC2 platform;
- performance evaluations through both repeatable simulation studies based on historical data and real experiments on Amazon EC2;

The remainder of the paper is organized as follows. We first model our problem in Section 2. In Section 3, we propose the base auto-scaling policies using heterogeneous spot instances under hourly billed context. Section 4 explains the optimizations we proposed on the initial polices. Section 5 briefly introduces our prototype implementations. We present and analyze the results of the performance evaluations in Section 6 and discuss the related works in Section 7. Finally, we conclude the paper and vision our future work.

2. System model

For reader's convenience, the symbols used in this paper are listed in Table 1.

Table 1
List of symbols.

Symbol	Meaning
$\mathbf{T}$	The set of spot types
M_{min}	The minimum allowed resource margin of an instance
M_{def}	The default resource margin of an instance
Q	The quota for each spot group
R	The required resource capacity for the current load
F_{max}	The maximum allowed fault-tolerant level
f	The specified fault-tolerant level
O	The minimum percentage of on-demand resources in the provision
S	The maximum number of selected spot groups in the provision
r_o	The resource capacity provisioned by on-demand instances
s	The number of chosen spot groups
vm	The VM type
vm_o	The on-demand VM type
c_{vm}	The hourly on-demand cost of the *vm* type instance
$num(c, vm)$	The function returns the number of *vm* type instances required to satisfy resource capacity *c*
C_o	The hourly cost of provision in on-demand mode
tb_{vm}	The truthful bidding price of *vm* spot group
m	The dynamic resource margin of an instance

2.1. Auto-scaling system architecture

As illustrated in Fig. 2, our auto-scaling system provisions a single-tier (usually the application server tier) of an application using a mixture of on-demand instances and spot instances. The provisioned on-demand instances are homogeneous instances that are most cost-efficient regarding the application, while spot instances are heterogeneous.

Like other auto-scaling systems, our system is composed of the *monitoring* module, the *decision-making* module, and the *load balancer*. The monitoring module consists of multiple independent monitors that are responsible for fetching newest corresponding system information such as resource utilizations, request rates, spot market prices, and VMs' statuses into the system. The decision-making module then makes scaling decisions according to the obtained information based on the predefined strategies and policies when necessary. Since in our proposed system provisioned virtual cluster is heterogeneous, the load balancer should be able to distribute requests according to the capability of each attached VM. The algorithm we use in this case is *weighted round robin*.

The application hosted by the system should be stateless. This restriction does not reduce the applicability of the system as modern cloud applications are meant to de developed in a stateless way in order to realize high scalability and availability (Wilder, 2012). In addition, stateful applications can be easily transformed into stateless services using various means, e.g., storing the session data in a separated memcache cluster.

2.2. Fault-tolerant mechanism

Suppose there are sufficient temporal gaps between price variation events of various types of spot VMs, increasing spot heterogeneity in provision can improve robustness. As illustrated in Fig. 3(a), the application is fully provisioned using 40 *m3.medium* spot VMs only, which may lead it to losing 100% of its capacity when *m3.medium*'s market price go beyond the

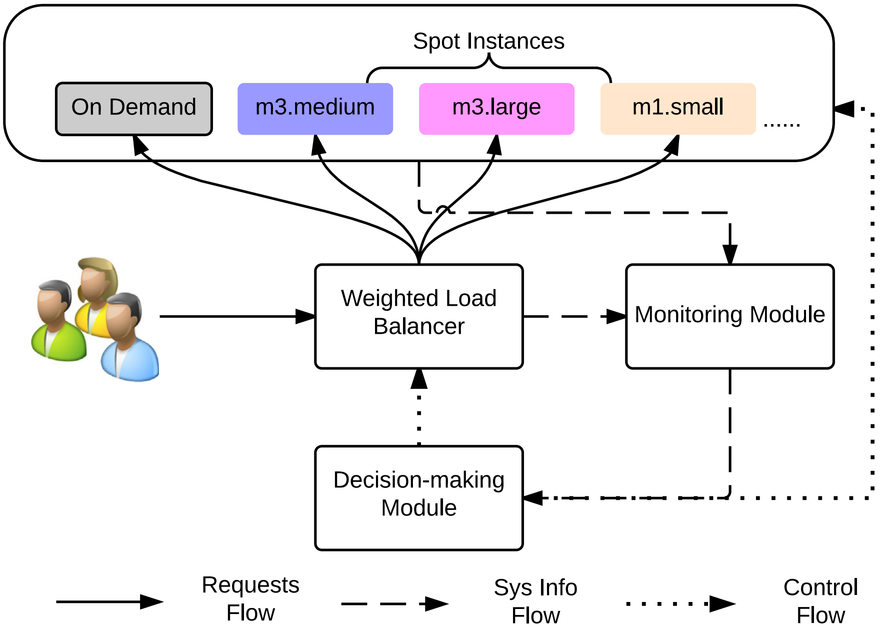


Fig. 2. Proposed auto-scaling system architecture.

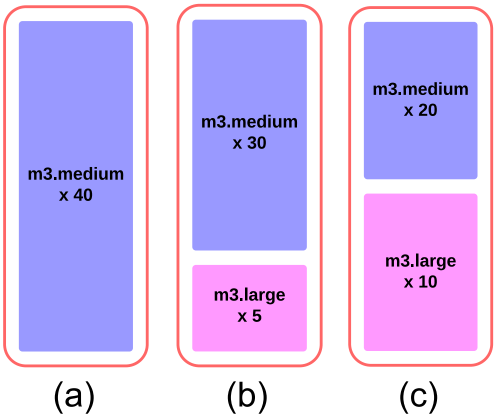


Fig. 3. Naive provisioning using spot instances.[1]

bidding price. By respectively provisioning 75% and 25% of the total required capacity using 30 *m3.medium* and 5[2] *m3.large* spot VMs in Fig. 3(b), it will lose at most 75% of its processing capacity when the price of either chosen type rises above the bidding price. Furthermore, if it is provisioned with equal capacity using the two types of spot VMs, like in Fig. 3(c), termination of the either type of VMs will only cause it to lose 50% of its capacity.

This is still unsatisfactory as we demand application performance to be intact even when unexpected termination happens. Simply, the solution is to further over-provision the same amount of capacity using another spot type, as the example illustrated in Fig. 4(b), it can be 50% of the required capacity provisioned using 9 *c3.large* instances. In this way, the application is now able to tolerate the termination of any involving type of VMs and remain fully provisioned. After detection of the termination, the scaling system can either provision the application using another type of spot VMs or switch to on-demand instances. Application performance is unlikely to be affected if there is no other termination happens before the scaling operation that repairs the provision fully completes.

However, it takes quite a long time to acquire and boot a VM (around 2 min for on-demand instances and 12 min for spot instances (Ming and Humphrey, 2012)). Hence, there is substantial possibility that another type of spot VMs could be terminated within this time window. To counter such situation, it requires further over-provision the application using extra spot types. We define the ***fault–tolerant level*** of our auto-scaling system as the maximum number of spot types that can be unexpectedly terminated without affecting application performance before its provision can be fully recovered. Fig. 4 respectively shows the provision examples that comply with fault-tolerant level zero, one, two, and three in our definition with each spot type provisioning 50% of the required capacity.

Note that setting fault-tolerant level to zero is usually not recommended. Though using multiple types of spot instances confines amount of resource loss when failures happen, with no over-provision to compensate resource loss, it may frequently cause performance degradations as failure probability becomes higher when more types of spot instances are involved.

2.3. Reliability and cost efficiency

Though the provisions shown in Fig. 4(b)–(d) successfully increase reliability of the application, they are not cost-efficient. The three provisions respectively over-provision 50%, 100%, and 150% of resources required by the application, which greatly diminishes the cost saving of using spot instances.

One possible improvement is to provision the application using more number of spot types. The illustrative provisions in Fig. 5 employ two more spot types than that are used in Fig. 4 to reach the corresponding fault-tolerant levels. As the result, total over-provisioned capacities for the three cases are reduced to 25%, 50%, and 75%. Though the provisions now might become more volatile

[1] The outer rectangles in Figs. 3–6 stand for the minimum amount of capacity required to process the current workload. Its value is dynamic and proportional to the changing workload so as the amount of redundancy for fault-tolerance.

[2] According to Amazon's specification, the capacity of 1 m3.large instance is equal to the capacity of 2 m3.medium instances.

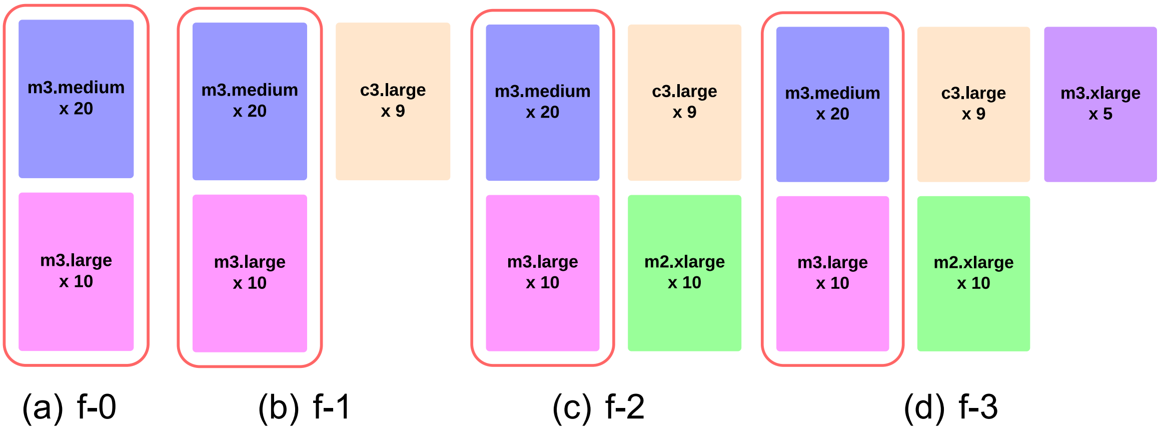


Fig. 4. Provisioning for different fault-tolerant levels.

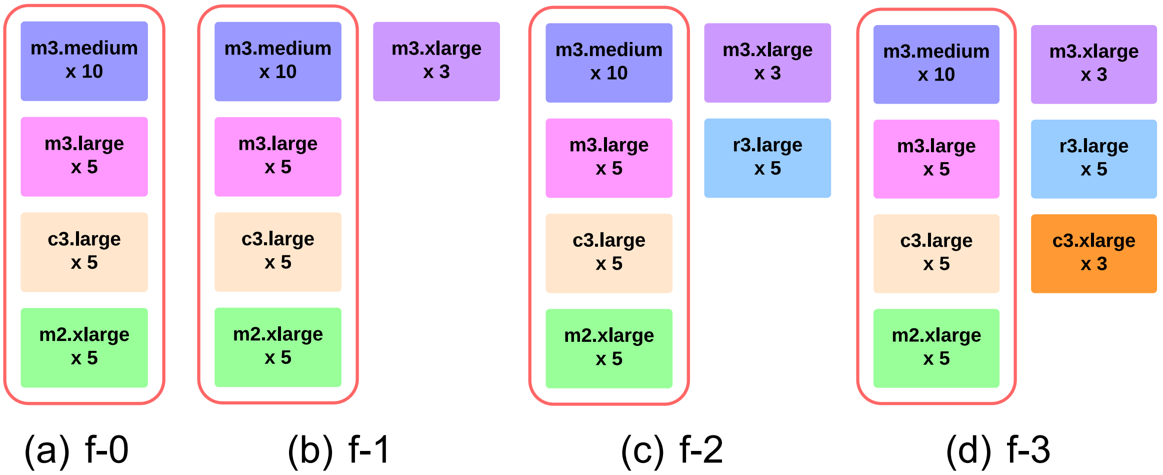


Fig. 5. Provisioning for different fault-tolerant levels using 2 more spot types.

with more types of spot VMs involved, the increased risk is manageable by the fault-tolerant mechanism with over-provision.

To reduce over-provision, the other choice is to provision the application with a mixture of on-demand instances and spot instances. Like the demonstrations shown in Fig. 6, there are now only 20%, 40%, and 60% over-provisioned capacities if 20% of the required resource capacity is provisioned by on-demand instances. Moreover, using on-demand resources also further confines amount of capacity that could be lost unexpectedly, thus, improving robustness. On the other hand, this method incurs more financial cost.

We define total capacity that is provisioned by the same type of spot VMs as a ***Spot Group***. In addition to that, we give definition to ***Quota*** (Q), which is the capacity each spot group needs to provision given the capacity provisioned by on-demand resources (r_o) and the fault-tolerant level (f). It is calculated as

$$Q = \frac{R - r_o}{s - f} \tag{1}$$

where R represents the required capacity for the current load and s denotes the number of chosen spot types. The minimum amount of capacity that is required to over-provision then can be calculated as $Q * f$.

We call a provision is ***safe*** if the provisioned capacity of each spot group is larger than Q. Hence, the problem of scaling web applications using heterogeneous spot VMs is transformed to dynamically selecting spot VM types and provisioning corresponding spot and on-demand VMs to keep the provision in safe state with minimum cost when the application workload increases, and timely deprovisioning various types of VMs when they are no longer needed.

3. Scaling policies

Based on the previous fault-tolerant model, we propose cost-efficient auto-scaling policies that comply with the defined fault-tolerant semantics for hourly billed cloud market like Amazon EC2.

3.1. Capacity estimation and load balancing

Our auto-scaling system is aware of multiple resource dimensions (such as CPU, Memory, Network, and Disk I/O). It needs the profile of the target application regarding its average resource consumption for all the considered dimensions. Currently, the profiling needs to be performed offline, but our approach is open to integrate dynamic online profiling into it.

With the profile, the system is able to estimate the processing capability of each spot type under the context of the scaling application. Based on that, it can easily determine how to distribute incoming requests to the heterogeneous VMs to balance their loads. In addition, the estimated capabilities are used in the calculation of scaling plans as well.

3.2. Spot mode and on-demand mode

Our scaling system runs interchangeably in ***Spot Mode*** and ***On−DemandMode***. Spot Mode provisions' application in the way is explained in Section 2.3. In Spot Mode, user needs to specify the minimum percentage of required resources provisioned by on-demand instances, symbolized as *O*. He can also set a limit on the number of selected spot groups in provision, denoted as *S*. To define these parameters, users can utilize the simulation tool implemented by us (described in Section 5) to find the optimal configurations according to the recent spot market history without running real tests on the cloud. Furthermore, these parameters can be dynamically adjusted using machine learning technologies. We leave this as our future work. In On-Demand Mode, application is fully provisioned by on-demand instances without over-provision. Switches between modes are dynamically triggered by the scaling policies detailed in the following sections.

3.3. Truthful bidding prices

Bidding truthfully means the participant in an auction always bids the maximum price he is willing to pay. In order to guarantee cost-efficiency, truthful bidding price for each VM type in our policies is calculated dynamically according to real-time workload and provision. Before computing them, we first calculate the hourly baseline cost if the application is provisioned in On-Demand Mode, which can be represented as

$$C_o = num(R, vm_o) * c_{vm_o} \tag{2}$$

where function $num(R, vm_o)$ returns the minimum number of instances of on-demand VM type required to process the current workload. c_{vm_o} is the on-demand hourly price of on-demand instance type. Then truthful bidding price of spot type *vm* is derived as follows:

$$tb_{vm} = \frac{C_o - num(r_o, vm_o) * c_{vm_o}}{s * num(Q, vm)} \tag{3}$$

where $num(r_o, vm_o)$ and $num(Q, vm)$ are interpreted similar to $num(R, vm_o)$ in Eq. (2).

This ensures that even in the worst situation that all chosen spot types' market prices are equal to their corresponding truthful bidding prices, the total hourly cost of the provision will not exceed that in On-Demand Mode.

Algorithm 1. Find new provision when the system needs to scale up.

```
Input: $R$: the current workload
Input: $n_c$: the number of on-demand VMs in current provision
Input: $vm_o$: the on demand $vm$ type
Input: $O$: the minimum percentage of on-demand resources
Output: target_provision
$min\_vm_o \leftarrow \mathbf{max}(n_c, num(R*O, vm_o))$;
$max\_vm_o \leftarrow num(R, vm_o)$;
candidate_set ← call Algorithm 2 for each integer $n$ in
    $[min\_vm_o, max\_vm_o]$;
return on-demand provision if candidate_set is empty
otherwise the provision with minimum cost in
    candidate_set;
```

Algorithm 2. Find provision given the number of on-demand instances.

```
Input: $n$: the number of on-demand VMs
Input: $g_c$: the set of spot groups in current provision
Input: $vm_o$: the on-demand $vm$ type
Input: $f$: the fault-tolerant level
Input: $\mathbf{T}$: the set of spot types
Input: $S$: the maximum number of chosen spot groups
Output: new_provision
$min\_groups \leftarrow \mathbf{max}(|g_c|, f+1)$;
$max\_groups \leftarrow \mathbf{min}(|\mathbf{T}|, S)$;
if $max\_groups < min\_groups$ then
    provision not found;
end
else
    for $s$ from $min\_groups$ to $max\_groups$ do
        $p \leftarrow p \cup (vm_o, n)$;
        compute $Q$ using Eq. (1);
        compute $tb_{vm}$ for each $vm$ in $\mathbf{T}$;
        $p \leftarrow p \cup g_c$;
        groups ← each group not in $g_o$ and whose $tb_{vm}$ is
         higher than market price;
        $k \leftarrow s - |g_c|$;
        if $|groups| \geq k$ then
            $p \leftarrow p \cup$ top k cheapest groups in groups;
            provisions ← provisions ∪ $p$;
        end
    end
end
return the cheapest provision in provisions;
```

Algorithm 3. Find target provision when the billing hour of one on-demand instance is about to end.

```
Input: $R$: the current workload
Input: $n_c$: the number of on-demand instances in current
provision
Input: $vm_o$: the on-demand $vm$ type
Input: $O$: the minimum percentage of on-demand
resources
Output: target_provision
if $n_c \leq num(R*O, vm_o)$ then
    provision not found;
end
else
    $p_1 \leftarrow$ call Algorithm2 with $n_c$;
    $p_2 \leftarrow$ call Algorithm2 with $n_c - 1$;
    return on − demand provision if neither $p_1$ nor $p_2$ is
    found otherwise either provision that is cheaper;
end
```

3.4. Scaling up policy

Scaling up policy is called when some instances are terminated unexpectedly or the current provision cannot satisfy resource requirement of the application. By resource requirement, in Spot Mode, it means the provision should be ***safe*** under the current workload, which is defined in Section 2.3. While in On-Demand Mode, it only requires the resource capacity of the provision to exceed the resource needs of the current workload.

Algorithm 1 is used to find the ideal new provision when the system needs to scale up. To avoid frequent drastic changes, the

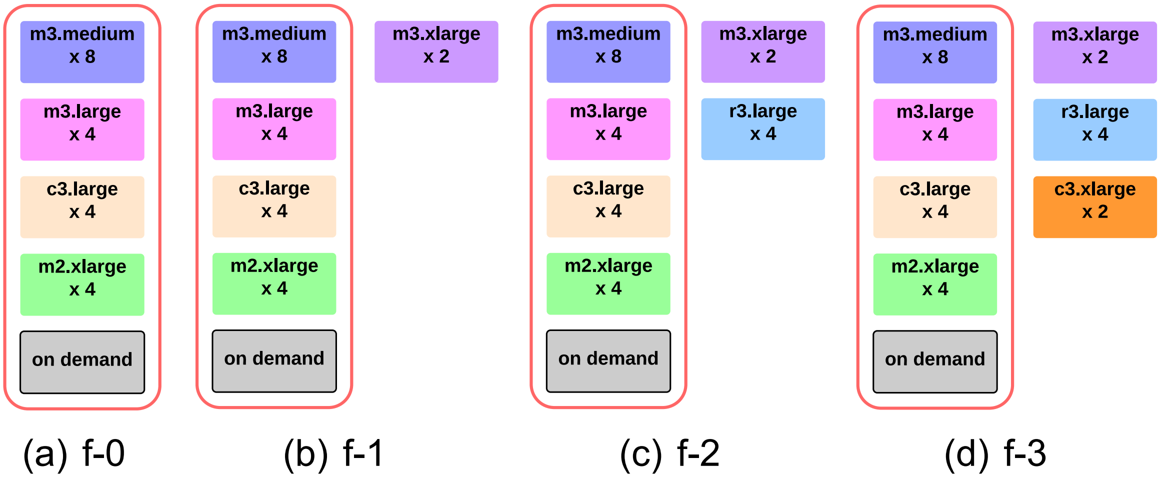


Fig. 6. Provisioning for different fault-tolerant levels using mixture of on-demand and spot instances.

algorithm only provisions VMs incrementally. As shown by line 1 in Algorithm 1, it limits the number of provisioned on-demand instances to be at least its current number. For each valid number of on-demand instances, it calls Algorithm 2 to find the corresponding best provision among provisions with various combinations of spot groups. Similarly, in Algorithm 2 (line 11), it retains the spot groups chosen by the current provision and only incrementally adds new groups according to their cost-efficiency (line 15). If there is no valid provision found, the system switches to on-demand mode.

After the target provision is found, the system compares it with the current provision and then contacts the cloud provider through its API to provision the corresponding types of VMs that are in short.

In the worst case, the time complexity of the scaling up policy is $O(N*S*|\mathbf{T}|)$ where N is the number of on demand instances required to provision the current workload in on demand mode, S denotes the maximum number of chosen spot groups, and $|\mathbf{T}|$ is the number of spot types considered. Since the parameters are all small integers, the computation overhead of the algorithm is acceptable in an online decision making scenario.

3.5. *Scaling down policy*

Since each instance is billed hourly, it is unwise to shut down one instance before its current billing hour matures. We therefore put the decision of whether each instance should be terminated or not at the end of their billing hours. The specific decision algorithms are different for on-demand instances and spot instances.

3.5.1. *Policy for on-demand instances*

When one on-demand instance is at the end of its billing hour, we not only need to decide whether the instance should be shut down, but also have to make changes to the spot groups if necessary. The summarized policy is abstracted in Algorithm 3. The algorithm first checks whether enough on-demand instances are provisioned to satisfy the on-demand capacity limit (line 1 and line 2). If there are sufficient on-demand instances, it endeavors to find the most cost-efficient provisions with and without the on-demand instance by calling Algorithm 2 (line 5 and line 6). Suppose the current provision is in On-Demand Mode and no provision is found without the on-demand instance, the provision will remain in On-Demand Mode. Otherwise, if a new provision is found without the current instance, the policy switches the provision to Spot Mode. In the case that the current provision is already in Spot Mode, it picks whichever provision that incurs lower hourly cost.

3.5.2. *Policy for spot instances*

When dealing with a spot instance whose billing period is ending, in the base policy, we simply shut down the instance when the corresponding spot quota Q can be satisfied without it. Thereafter, the policy will evolve with the introduced optimizations in Section 4.

3.6. *Spot groups removal policy*

Note that in both scaling up and down policies, we forbid removing selected spot groups from provision. Instead, we evict a chosen spot group when any spot instances of such type is terminated by the provider. Since bidding price of each instance is calculated dynamically, instances within the same spot group may be bid at different prices. This could cause some instances to remain alive even after the corresponding spot groups are removed from provision. We call the instances that are running but do not belong to any group ***orphans***. Though orphan instances are still in production, they are not considered as a part of the provision according to the fault-tolerant semantics when making scaling decisions. In the base policies, although they will not be shut down until their billing hour ends, extra instances still need to be launched to comply with the fault-tolerant semantics, which causes resource waste. This drawback is addressed by the introduced optimizations in the following section.

4. Optimizations

We have made several optimizations on the above proposed base policies to further improve cost-efficiency and reliability of the system.

4.1. *Bidding strategy*

In the scaling policies, spot groups are bid at truthful bidding prices calculated by Eq. (3) due to cost-efficiency concern. While focusing on robustness, the system can employ a different strategy to bid higher so as to grasp spot instances as long as possible.

4.1.1. *Actual bidding strategies*

There are two actual bidding strategies, namely truthful bidding strategy and on-demand price bidding strategy embedded in the system.

- *Truthful bidding strategy*: the system always bids the truthful bidding price calculated by Eq. (3) when new spot instances are launched. Since partial billing hours ended by cloud provider are free of charge, cloud users can save money by letting cloud provider terminate their spot instances once their market prices exceed the corresponding truthful bidding prices. On the other hand, it leads to more unexpected terminations.
- *On-demand price bidding strategy*: the system always bids the on-demand price of the corresponding spot type whenever trying to obtain new spot instances. This strategy will cost cloud users more money but provides a higher level of protection against unexpected terminations.

4.1.2. *Revised spot groups' removal policy*

In the base policies, less cost-efficient spot groups could remain in provision for a long time unless some of their instances are terminated by provider. When the actual bids are higher than the truthful bidding prices, the situation could become worse. Instead of just relying on provider terminating uneconomical spot groups, the revised policy actively inspects whether market prices of some spot groups have exceeded their corresponding truthful bidding prices and remove them from the provision. In the meantime, for spot groups whose market prices are still below their truthful bidding prices, it looks for chance to replace them by more economical spot groups that have not been selected. To minimize disturbance to provision, such operations should be conducted in a long interval, such as every 30 min in our implementation. Members of removed or replaced spot groups become orphans.

4.2. *Utilizing orphans*

After removing or replacing some spot groups, if the system simply lets members of these spot groups become orphans and immediately start instances of newly chosen spot groups, the stability of provision will be affected. Furthermore, as orphans are not considered as valid capacity in the base polices, during the transition period, it has to provision more resources than necessary, which results in monetary waste.

To alleviate this problem, we aim to utilize as many orphans in provision as possible to deter the time to provision new VMs. As a result, resource waste can be reduced and cost-efficiency is improved.

We modify the proposed fault-tolerant model to allow a spot group temporarily accept instances that are heterogeneous to the spot group type under certain conditions. Fig. 7 illustrates such provision. In Fig. 7 (a), the *m1.small* group does not have sufficient instances to satisfy its quota. Instead of launching 2 new *m1.small* spot instances, the policy now temporarily moves the available orphan, one *m1.medium* instance, to *m1.small* group to compensate the deficiency of its quota. Even though *m1.small* group becomes heterogeneous in this case, it does not violate the fault-tolerant semantics as losing any type of spot instances will not influence the application performance. However, in some situations, heterogeneity in spot groups could cause violation of the fault-tolerant semantics, for example, there might be case that three *m1.medium* orphans are spread across three spot groups and the total capacity of the three instances exceeds the spot quota. Then losing the three *m1.medium* instances will violate the fault-tolerant semantics. Fortunately, such cases are very rare as orphans are usually small in numbers and are expected to be shut down in a short time.

With this relaxation of the fault-tolerant model, the previous scaling up and scaling down policies need to be revised to efficiently utilize capacity of orphans.

4.2.1. *Revised scaling up policy*

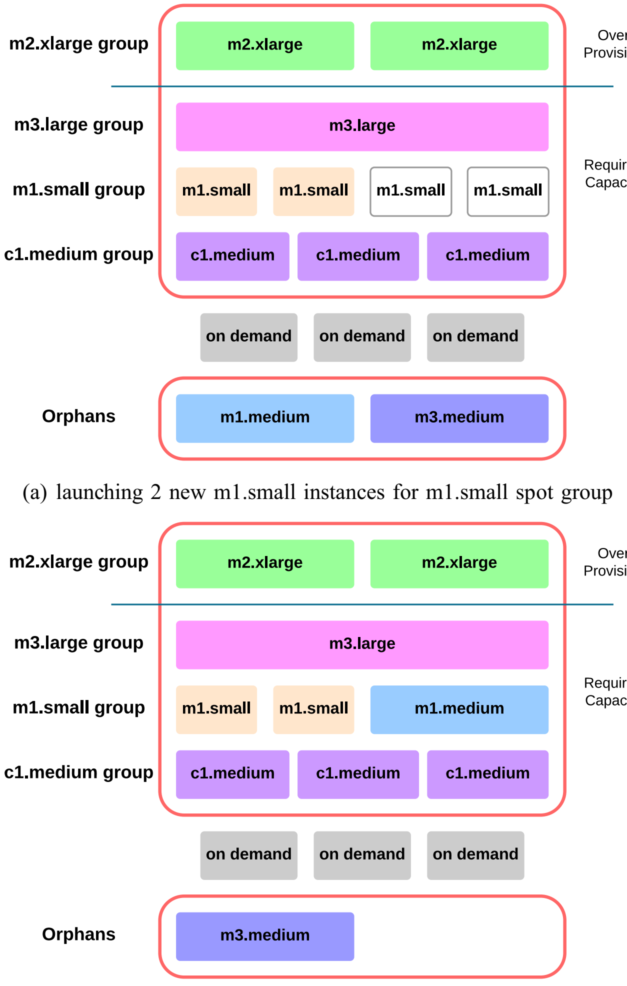
The new scaling up policy uses the same algorithm (Algorithm 1) to find the target provision. However, instead of simply launching instances to reach the target provision, the new policies take a deeper thought whether it can utilize existing orphans to meet the quota requirements in the target provision.

The new policy first checks whether the target provision chooses new spot groups. If there are orphans whose types are the same to any newly chosen groups, lying either within orphan queue or other spot groups, they are immediately moved to the corresponding new spot groups. After that, the policies endeavor to insert non-utilized orphans from the orphan queue into spot groups that have not met their quota requirement. If all the orphans have been utilized and some groups still cannot satisfy their quota, new spot instances of the corresponding types then will be launched.

4.2.2. *Revised scaling down policy*

Regarding policy for on-demand instances that are close to their billing hour, the new policy utilizes the same mechanism in the revised scaling up policy to provision any changes between the current provision and the target provision.

For spot scaling down policy, if the spot instance is in orphan queue, it is immediately shut down. Suppose it is within the spot group of the same type, it is shut down when the spot quota can



(a) launching 2 new m1.small instances for m1.small spot group

(b) using one m1.medium orphan to temporarily substitute 2 m1.small instances

Fig. 7. Provisioning with orphans under fault-tolerant level one.

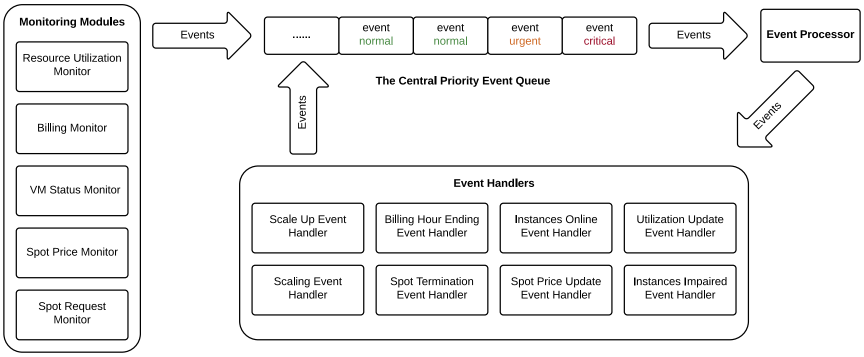

Monitoring Modules
Resource Utilization Monitor
Billing Monitor
VM Status Monitor
Spot Price Monitor
Spot Request Monitor
Events
......
event normal
event normal
event urgent
event critical
Events
Event Processor
The Central Priority Event Queue
Events
Events
Event Handlers
Scale Up Event Handler
Billing Hour Ending Event Handler
Instances Online Event Handler
Utilization Update Event Handler
Scaling Event Handler
Spot Termination Event Handler
Spot Price Update Event Handler
Instances Impaired Event Handler

Fig. 8. Components of the implemented auto-scaling system.

be satisfied without it. In the case that the instance is an orphan within other spot group, the new policy shuts down the instance and in the meantime starts certain number of spot instances of the spot group type to compensate the capacity loss.

4.3. *Reducing resource margin*

For applications running on traditional auto-scaling platform, administrator usually leaves a margin at each instance to handle short-term workload surge in order to buy time for booting up new instances. This margin empirically ranges from 20 to 25% of the instance's capacity.

With over-provision already in place in our system, this margin can be reduced under Spot Mode provision. We devise a mechanism that dynamically changes the margin according to the current fault-tolerant level. Since higher fault-tolerant level leads to more over-provision, we can be more aggressive in reducing the margin of each instance. In detail, the dynamic margin is determined by the formula:

$$m = \frac{M_{def} - M_{min}}{F_{max}} * f + M_{min} \tag{4}$$

where M_{min} means the minimum allowed margin, e.g., 10%, M_{def} is the default margin used without dynamic margin reduction, e.g., 25%, and F_{max} is the maximum allowed fault-tolerant level.

5. Implementation

We implemented a prototype of the proposed auto-scaling system on Amazon EC2 platform using Java, the components of which are illustrated in Fig. 8. It employs an event-driven architecture with the monitoring modules continuously generating events according to newly obtained information, and the central processor consuming events one by one. Monitoring modules produce and insert corresponding events with various critical levels into the central priority event queue. They include the *resource utilization monitors* that watch all dimensions of resource consumption of running instances, the *billing monitor* that gazes billing hour of each requested VM, the *VM status monitor* that reminds the system when some instances are online or offline, the *spot price monitor* that records newest spot market prices for each considered spot type, and *the spot request monitor* that surveillances any unexpected spot termination. On the other side, the central event processor fetches events from the event queue and assigns them to the corresponding event handlers that realize the proposed policies to make scaling decisions or perform scaling actions.

The prototype implementation provides a general interface for users to plug different load balancer solutions into the auto-scaling system. In our case, we use *HAProxy* with weighted round robin algorithm. It also offers the interface to allow users to automatically customize configurations of VMs according to their own available resources after they have been booted.

For quick concept validation and repeatable evaluation of the proposed auto-scaling policies, we created a simulation version of the system. The same code base is transplanted onto CloudSim (Calheiros et al., 2011) toolkit which provides the underlying simulated cloud environment. Assuming bids from user impose negligible influence on market prices, the simulation tool is able to provide quick and economical validation of the proposed polices using historical data of the application and the spot market as input.

For more details about the implementation, refer to the released code.[3]

6. Performance evaluation

6.1. *Simulation experiments*

As stated in Section 5, to allow repeatable evaluation, we developed a simulation version of the system that allows us to compare the performances of different configurations and policies using traces from real applications and spot markets.

6.1.1. *Simulation settings*

We use one week trace of 10% English Wikipedia requests from September 19th 2007 to September 26th 2007 as the workload (van Baaren, 2015; Urdaneta et al., 2009), which is depicted in Fig. 9. Note that our approach is general purpose and can be applied to any workload, as the proposed system does not make assumptions on the workload and is fully reactive. We adopt the Wikipedia workload in experiments because it reveals significant variations that can trigger frequent scaling operations to let us observe the behavior of our system. We believe one week trace is enough for the purpose of our experiments, as it gives the system ample opportunities to exercise the scaling policies. In addition, as reported by Eldin et al. (2014), the Wikipedia workload revealed strong weekly pattern with only gradual changes in amplitude, level, and shapes.

We consider 13 spot types in Amazon EC2. Their spot prices are simulated according to one week Amazon's spot prices history from March 2nd 2015 18:00:00 GMT in the relatively busy ***us*−*east*** region. The involving spot types and their corresponding history market prices are illustrated in Fig. 1.

[3] https://github.com/quchenhao/spot-auto-scaling

We set requests timeout at 30 s. In addition, we respectively set minimum allowed resource margin (M_{min}) and default resource margin (M_{def}) at 10% and 25%. We found out that *c3.large* instance is the most cost-efficient type for the wikipedia application based on a small scale resource profiling test of the Wikibench application (van Baaren, 2009) on Amazon EC2 and the resource specifications of each instance type released by Amazon. It is selected to provision all the on-demand resources in the experiments. All simulation experiments start with 5 *c3.large* on-demand instances. Length of simulated requests is generated following a pseudo-Gaussian distribution[4] with mean of 0.07 ECU[5] and standard deviation of 0.005 ECU so that different tests using the same random seed are receiving exactly the same workload. The VM start up, shut down, and spot requesting delays are generated in the same way using pseudo-Gaussian distribution. The means of the above three distributions are respectively 100, 100, 550 s, and the standard deviations are set at 20, 20, 50 s. The test results are deterministic and repeatable on the same machine.

We tested our scaling policies with various fault-tolerant levels and different least amounts of on-demand resources, which are represented respectively as "f-*x*" and "*y*% on-demand" in the results. We also tested the polices using the two embedded bidding strategies and static/dynamic resource margins.

We concentrate on two metrics, real-time response time of requests (average response time per second reported) and total cost of instances, in all the experiments.

6.1.2. Benchmarks

We compare our scaling policies with two benchmarks:

- *On-demand auto-scaling*: This benchmark only utilizes on-demand instances. It is implemented by restricting the auto-scaling system always in On-Demand Mode.
- *One spot type auto-scaling*: The auto-scaling policies used in this benchmark, like the proposed policies, provision a mixture of on-demand resources and spot resources. The benchmark also has a limit on minimum amount of on-demand resources provisioned. However, for spot instances, it only provisions one spot group that is the most cost-efficient at the moment without over-provision. If the provisioned spot instances are terminated, a new spot group then is selected and provisioned. Suppose a more economic spot group is found, the old spot group is gradually replaced by the new one. It is implemented by setting fault-tolerant level to zero and limiting at most one spot group can be provisioned.

6.1.3. Response time

Figs. 10–13 respectively depict real-time average response time of requests using on-demand, one spot, and our approach with truthful bidding strategy and dynamic resource margin. From the results, the on-demand auto-scaling produced smooth response time all along the experimental duration except for a peak that was caused by the corresponding peak in the workload. All experiments employing one spot type auto-scaling experienced periods of request timeouts caused by termination of spot instances, and only increasing the amount of on-demand resources cloud not improve the situation. While our approach greatly reduced such unavailability of service even using f-0 with no over-provision of resources. By using f-1, we were able to completely eliminate the timeouts under the recorded spot market traces. We omit the results for tests using f-2 and f-3 as they reveal similar results as Fig. 13.

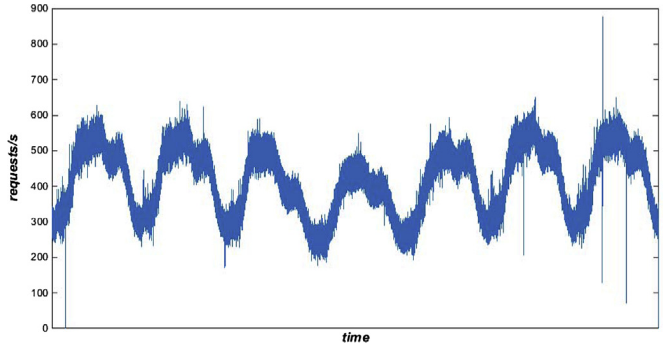


Fig. 9. The English Wikipedia workload from September 19th 2009 to September 26th 2009.

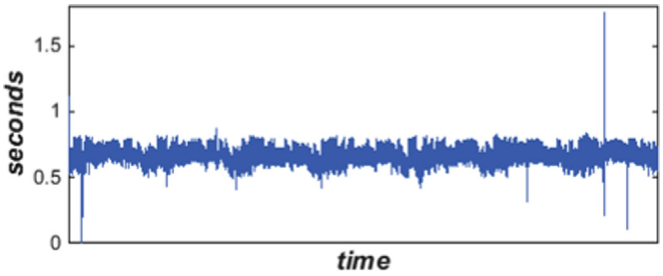


Fig. 10. Response time for on-demand auto-scaling.

To show the effect of different bidding strategies, we compare the response time results of one spot type auto-scaling using the two proposed bidding strategies as they reveal the most significant difference. As Figs. 11 and 14 present, it is obvious that service availability can be much improved with higher bidding prices using one spot type auto-scaling. On the other hand, the remaining timeouts also indicate that increasing bidding prices alone is not enough to guarantee high availability.

6.1.4. Cost

Table 2 lists the total costs produced by all the experiments. Comparing to the cost of on-demand auto-scaling, we managed to gain significant cost saving using all other configurations. Tests using one spot type auto-scaling with 0% on-demand resources realized the most cost saving up to 80.87% regardless of its availability issue.

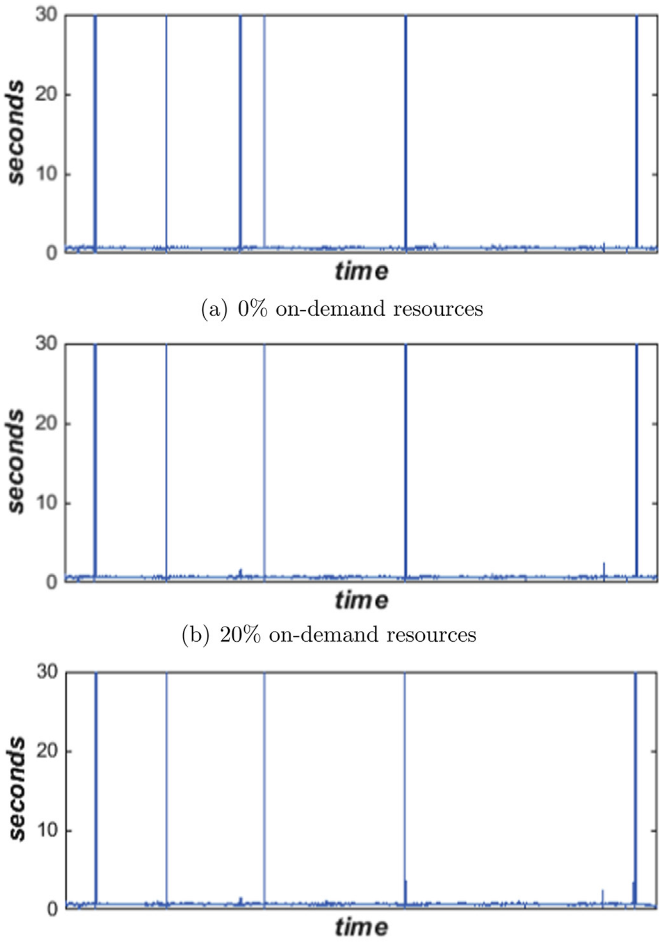
The results show that the amount of on-demand resources has a significant influence on cost saving. It also can be noted that higher fault-tolerant level incurs extra cost. Though optimal configuration of fault-tolerant level is always application specific, according to our results, configuration using f-1 with 0% on-demand resource is the best choice under current market situation in regards of both financial cost and service availability.

The resulted cost differences caused by different bidding strategies are generally small. Therefore, it is better to bid higher to improve availability if user's bidding has negligible impact on the market price.

As dynamic resource margin is only applicable when application is over-provisioned, we give the results for tests using dynamic resource margin when fault-tolerant level is higher than zero. According to the results, dynamic resource margin can bring extra cost saving and the amount of cost saving increases when more over-provision is necessary (i.e., higher fault-tolerant level). Though the

[4] Since Wikipedia is serving mostly the same type of requests – page view, the time taken to process each request is also likely to fall in a certain interval. To coarsely model such behavior, we utilize Gaussian distribution. Other distributions with small head and tail can serve the same purpose as well.

[5] It means the request takes 70 ms to finish if it is computed by the VM equipped with vCPU as powerful as 1 Elastic Computing Unit (ECU).

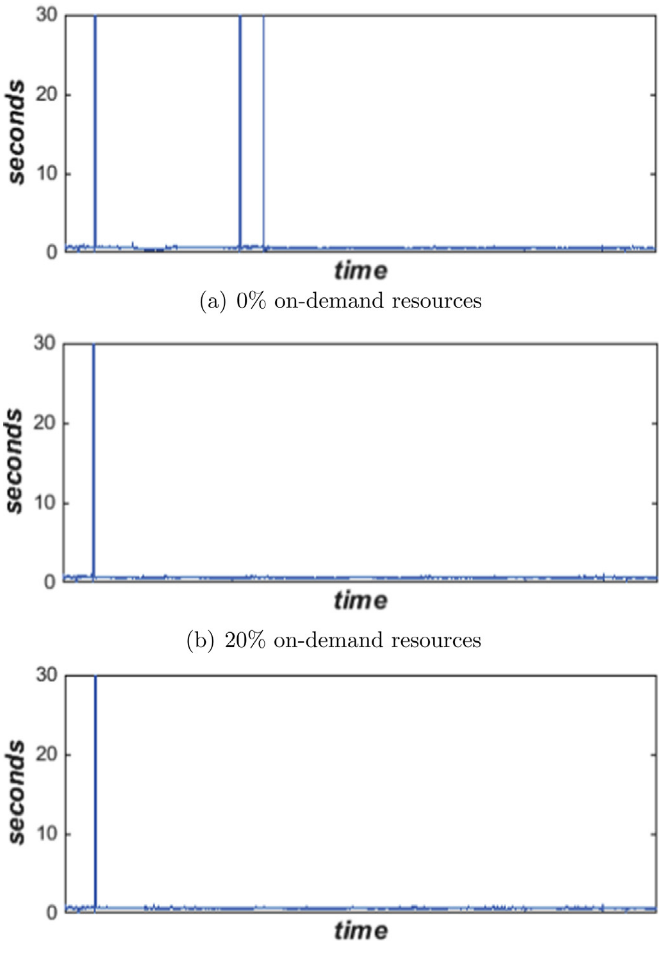


(a) 0% on-demand resources

(b) 20% on-demand resources

(c) 40% on-demand resources

Fig. 11. Response time of one spot type auto-scaling with various percentage of on-demand resources and truthful bidding strategy.



(a) 0% on-demand resources

(b) 20% on-demand resources

(c) 40% on-demand resources

Fig. 12. Response time of f-0 with various percentage of on-demand resources, truthful bidding strategy, and dynamic resource margin.

resulted cost saving is not significant, it is safely achieved without sacrificing availability and performance of the application.

6.2. Real experiments

We conducted two real tests on Amazon EC2 respectively using on-demand auto-scaling policies and the proposed auto-scaling policies with configuration of f-1 and 0% on-demand. Other parameters are defined the same to the simulation tests.

We set up the experimental environment to run the Wikibench (van Baaren, 2009) benchmark tool. The major advantage of this tool compared to other tools such as TPC-W, RUBiS, and CloudStone is that it is stateless, which is the characteristic of modern highly scalable cloud services (Wilder, 2012). The tool is composed of three components:

- a client driver that mimics clients by continuously sending requests to the application server according to the workload trace;
- a stateless application server installed with the Mediawiki application;
- a mysql database loaded with the English Wikipedia data by the date of January 3rd, 2008.

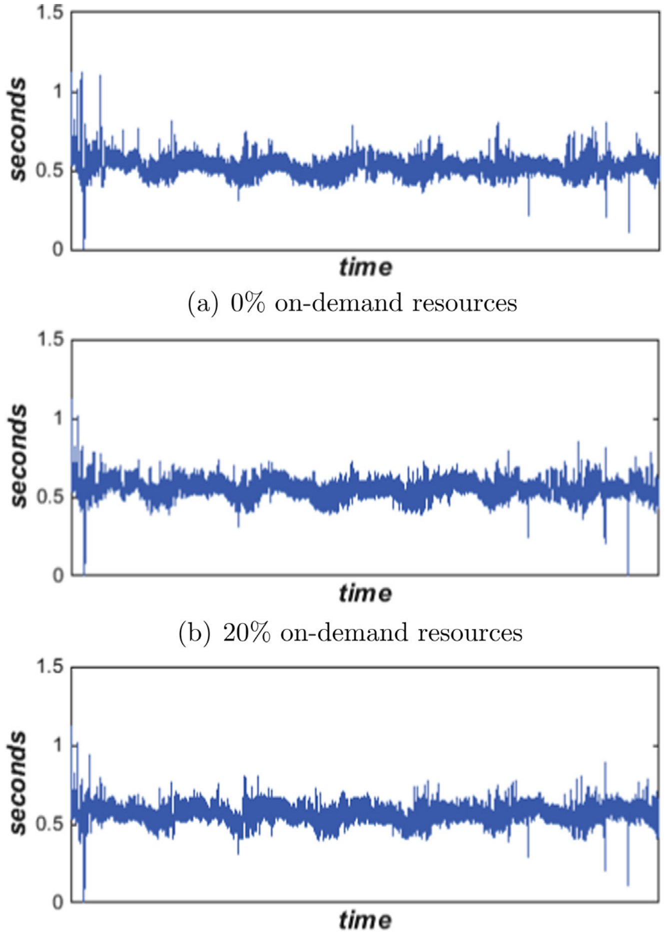
Our aim is to scale the application-tier. Thus, we inserted a HAProxy load balancer layer into the original architecture in order to let the client driver talk to a cluster of servers. The architecture of the testbed is illustrated in Fig. 15. We picked the first 3 days of the Wikipedia workload (van Baaren, 2015; Urdaneta et al., 2009) (Fig. 9) and scaled it down to half of its original rate as the workload for testing because Amazon limits the number of instances each account can launch.

The testing environment resided in Amazon ***us***−***east***−***1d*** zone which is in a relatively busy region with higher degree and frequency of price fluctuations. Regarding each component, we launched one *c4.large* instance acting as the client driver, one *m3.medium* instance running the HAProxy load balancer, and one *c4.2xlarge*[6] instance serving the mysql database requests. The auto-scaling system itself is running on a local desktop computer remotely in Melbourne. Before the tests, we profiled each component to make sure none of them become the bottleneck of the system.

The test using the proposed approach started at 3:30am September 9, 2015, Wednesday, US east time. Its testing period spanned across three busy weekdays from Wednesday to Friday.

Figs. 16 and 17 presents real-time response time results of the two experiments. Both results suffer from peaks of high response time. By studying the recorded log, we confirmed they were not caused by shortage of resources as resource utilizations of all the involving VMs were never beyond safe threshold during both tests. Various other reasons can be the culprits, such as cold cache,

[6] The 4th generation instances were introduced between the time we performed the simulations and the real experiments. To be consistent, we only consider the 13 spot types listed in Fig. 1 for both the simulations and the real experiments.

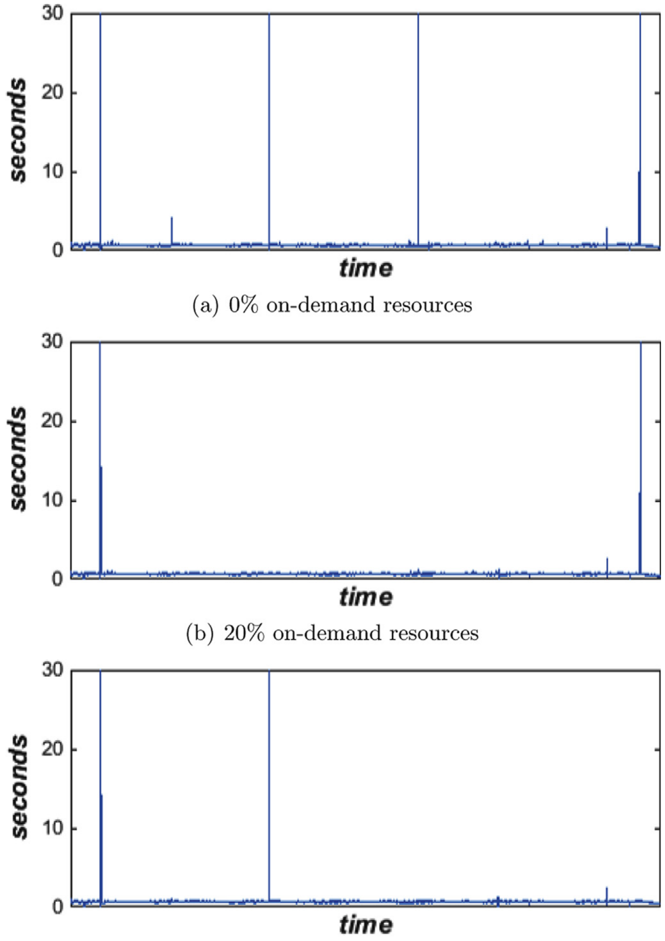


(a) 0% on-demand resources

(b) 20% on-demand resources

(c) 40% on-demand resources

Fig. 13. Response time of f-1 with various percentage of on-demand resources, truthful bidding strategy, and dynamic resource margin.



(a) 0% on-demand resources

(b) 20% on-demand resources

(c) 40% on-demand resources

Fig. 14. Response time of one spot type auto-scaling with various percentage of on-demand resources and on-demand bidding strategy.

short term network issues, interference from the shared virtualized environment, and garbage collection (Dean and Luiz, 2013). We encountered three unexpected terminations during the test of our approach. Thanks to the fault-tolerant mechanism and policies, we managed to avoid service interruption and performance degradation during those periods.

In addition, because resources are tighter in on-demand auto-scaling, it generally performs worse in response time compared to the proposed approach.

Regarding cost, we calculated the total cost of application servers in both experiments. Table 3 presents the results. The proposed approach reaches 70.07% cost saving.

6.3. Discussion

Even with high fault-tolerant level, the proposed approach cannot guarantee 100% availability, and no solution can ever manage to assure absolute service continuity due to the nature of spot market. What our system offers is a best effort to counter large scale surges of market prices of the selected spot types in a short time, which is highly unlikely under current market condition. In fact, we have not encountered any case that more than one spot group fail simultaneously during simulations, real experiments, and testing phases. However, market condition could change. Hence, application provider should adjust configuration of the auto-scaling system dynamically according to real-time volatility of the spot market. In addition, the nature of the application also affects the decision. If the application is availability-critical, higher fault-tolerant level is always desirable. Adversely, for some applications, such as analytical jobs, even one spot type auto-scaling is acceptable.

The presented results in Section 6 only indicates the cost saving potential of a certain application considering a selected set of spot types under the recorded spot market prices and workload traces. Thanks to the dynamic truthful bidding price mechanism, even in competitive market condition, we can ensure that the cost reduction gained by our approach will not vanish but only diminish. To reach more cost saving, the application provider can take into account a broader set of spot types, which is available in Amazon's offering.

To save cost and time for testing, application providers can tune the parameters of the auto-scaling system in a similar way as we did by first utilizing simulation for fast validation and then test the system in production environment.

There are also differences in price among the same spot types across different availability zones. It is trivial to extend the current fault-tolerant model to utilize spot groups from multiple availability zones. Currently, the auto-scaling system limits the selection of spot groups within the same availability zone due to charges for traffic across availability zones. If the application provider has already adopted a multi-availability-zone deployment, such extension is able to realize more cost saving.

The overhead of the auto-scaling system is negligible. As presented in Section 3, the time complexity of the scaling policies is not significant. The frequency that the scaling policies are called

Table 2
Total costs for experiments with various configurations.

Policies	Total cost (USD$)			
On-demand	116.34			
	Truthful bidding		**On-demand bidding**	
One spot with 0% **on-demand**	22.26		23.14	
One spot with 20% **on-demand**	46.50		47.33	
One spot with 40% **on-demand**	63.17		63.43	
f-0 **with** 0% **on-demand**	32.00		32.30	
f-0 **with** 20% **on-demand**	54.45		56.10	
f-0 **with** 40% **on-demand**	68.34		69.64	
	Static resource margin	**Dynamic resource margin**	**Static resource margin**	**Dynamic resource margin**
f-1 **with** 0% **on-demand**	41.57	39.32	43.17	41.66
f-1 **with** 20% **on-demand**	60.21	59.52	61.82	61.06
f-1 **with** 40% **on-demand**	72.09	72.55	72.96	73.08
f-2 **with** 0% **on-demand**	50.48	47.38	51.67	49.38
f-2 **with** 20% **on-demand**	67.72	65.52	68.71	66.09
f-2 **with** 40% **on-demand**	78.01	76.74	78.3	76.74
f-3 **with** 0% **on-demand**	67.87	62.61	68.79	61.50
f-3 **with** 20% **on-demand**	83.27	78.33	81.18	76.19
f-3 **with** 40% **on-demand**	89.86	85.57	88.09	84.46

depends on the monitoring interval and the frequency of price changes, which are at least in the scale of seconds.

7. Related work

7.1. Horizontally auto-scaling web applications

Horizontally auto-scaling web applications have been extensively studied and applied (Lorido-Botran et al., 2014). Basically, auto-scaling techniques for web applications can be classified into three categories: *reactive approaches*, *proactive approaches*, and *mixed approaches*. Reactive approaches scale applications in accordance of workload changes. Proactive approaches predict future workload and scale applications in advance. Mixed approaches can scale applications both reactively and proactively.

Most industry auto-scaling systems are reactive-based. Among them, the most frequently used service is Amazon's Auto Scaling Service (Amazon, 2015). It requires user to first create an auto-scaling group, which specifies the type of VMs and image to use when launching new instances. Then user should define his scaling policies as rules like “add 2 instances when CPU utilization is larger than 75%”. Another popular service is offered by RightScale. Their service is based on a voting mechanism that lets each running instance decide whether it is necessary to grow or shrink the size of the cluster based on their own condition (RightScale, 2015).

Other than just using simple rules to make scaling decisions, researchers have developed scaling systems based on formal models. These models aim to answer the question that how many resources are actually required to serve certain amount of incoming workload under QoS constraints. Such model can be simply obtained using profiling techniques as we did in this paper. Other commonly adopted approaches include queueing models (Urgaonkar et al., 2008; Jiang et al., 2013; Han et al., 2014; Gandhi et al., 2014, 2014; Salah et al., 2015) that either abstract the application as a set of parallel queues or a network of queues, and online learning approaches such as reinformacement learning (Dutreilh et al., 2011; Barrett et al., 2013; Xiangping et al., 2013).

Proactive auto-scaling is desirable because time taken to start and configure newly started VMs creates a resource gap when workload suddenly surges to the level beyond capability of the available resources. To satisfy strict SLA, sometimes it is necessary to provision enough resources before workload actually rises. As workloads of web applications usually reveal temporal patterns, accurate prediction of future workload is feasible using state-of-art time-series analysis and pattern recognition techniques. A lot of them have been applied to auto-scaling of web applications (Jiang et al., 2013;

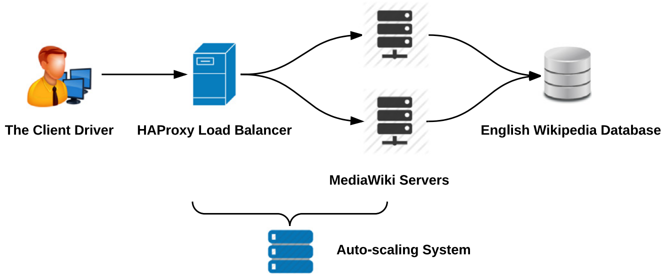


Fig. 15. The testbed architecture.

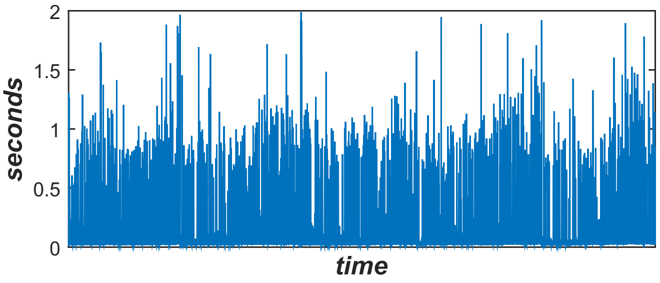


Fig. 16. Response time for on-demand auto-scaling on Amazon.

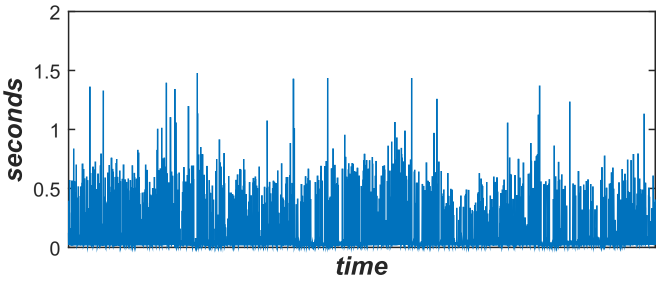


Fig. 17. Response time for spot auto-scaling on Amazon.

Table 3
Cost of the experiments.

Policies	**Total cost** (**USD$**)
On-demand	19.01
$ft-1$ **and** 0% **on-demand**	5.69

Roy et al., 2011; Jingqi et al., 2013; Caron et al., 2011; Islam et al., 2012; Wei et al., 2012; Herbst et al., 2014; Dutta et al., 2012).

Most auto-scaling systems only utilize homogeneous resources. While some, including our system, have explored using heterogeneous resources to provision web applications. Upendra et al. (2011), and Srirama and Ostovar (2014) adopt integer linear programming (ILP) to model the optimal heterogeneous resource configuration problem under SLA constraints. Fernandez et al. (2014) utilize tree paths to represent different combinations of heterogeneous resources and then search the tree to find the most suitable scaling plan according to user's SLA requirements.

Different from the above works, our objective goes beyond using minimum resources to provision the application. Instead, we want to devise fault-tolerant mechanism and auto-scaling policies that comply with the fault-tolerant semantics to reliably scale web applications on cheap spot instances. We believe the reviewed auto-scaling techniques are complementary to our approach. The proposed system can incorporate their resource estimation models, and workload prediction techniques as well.

7.2. *Application of spot instances*

There have been a lot of attempts to use spot instances to cut resource cost under various application context. Resource provision problems using spot instances have been studied for fault-tolerant applications (Costache et al., 2012; Binnig et al., 2015; Poola et al., 2014; Sifei et al., 2013; Voorsluys and Buyya, 2012; Changbing et al., 2014; Chohan et al.; Zafer et al., 2012; Hsuan-Yi and Simmhan, 2014; Subramanya et al., 2015) such as high performance computing, data analytics, MapReduce, and scientific workflow.

For these applications, the fault-tolerant mechanism is often built on checkpointing, replication, and migration. Multiple novel checkpointing mechanisms (Jangjaimon and Nian-Feng, 2015; Sangho et al., 2012; Jung et al., 2011) have been developed to allow these applications to harness the power of spot instances. SpotOn (Subramanya et al., 2015) combines multiple fault-tolerant mechanisms to increase the cost-efficiency and performance of batch processing applications running on spot instances.

Regarding web applications, Han et al. (2012) proposed a stochastic algorithm to plan future resource usage with a mixture of on-demand and spot instances. Except they only use homogeneous resources, their problem is also different to ours as they aim to plan the resource usage with the knowledge of the future while we provision resources dynamically. Mazzucco and Dumas (2011) also explored using a mixture of homogeneous on-demand instances and spot instances to provision web applications. Instead of building a reliable auto-scaling system, their target is to maximize web application provider's profit by using an admission control mechanism at the front end to dynamically adapt to sudden changes of available resources.

Sharma at al. proposed a derivative IaaS cloud platform based on spot instances called SpotCheck (Singh et al., 2014; Sharma et al., 2015). To transparently provide high availability on spot instances to end users, they incorporated technologies, such as nested virtualization, live VM migration, and time-bounded VM migration with memory checkpointing, to dynamically move users' VMs when underlying spot instances are available or revoked. Because of its transparency to end users, it is ideal for cloud brokers and large organizations with high resource demands. While our approach is lightweight and thus more suitable for small organizations who want to harness the power of spot instances by themselves. He et al. (2015) from the same group evaluated the ability of the approach to reliably run web applications on spot instances. Though they do not provision redundant capacity as we do, they reported non-negligible overhead incurred by nested virtulization. Their proposed system (Singh et al., 2014; Sharma et al., 2015; He et al., 2015) is able to preserve the memory state of the revoked spot VMs, which enables it to seamlessly host stateful applications. Though our approach requires the application to be stateless, this does not reduce its generality as highly scalable cloud applications are expected to be stateless (Wilder, 2012), and stateful applications can be easily turned into stateless by storing session information in a memory cache cluster (Wilder, 2012).

Their system relies on the termination warnings issued by existing providers (Amazon) to be able to conduct migrations in time. Our approach is capable of operating in possible future spot markets that do not provide termination warnings.

Recently, Amazon EC2 introduced a new feature, called Spot Fleet API (https://aws.amazon.com/blogs/aws/ new-ec2-spot-instance-termination-notices/). It allows user to bid for a fixed amount of capacity possibly constituted by instances of different spot types. It continuously and automatically provisions the capacity using the combination of instances that incurs the lowest cost. However, as its provision decision ignores reliability, it is not suitable to provision web applications.

8. Conclusions and future work

In this paper, we explored how to reliably and cost-efficiently auto-scale web applications using a mixture of on-demand and heterogeneous spot instances. We first proposed a fault-tolerant mechanism that can handle unexpected spot terminations using heterogeneous spot instances and over-provision. We then devised novel cost-efficient auto-scaling policies that comply with the defined fault-tolerant semantics for hourly-billed cloud market. We implemented a prototype of the proposed auto-scaling system on Amazon EC2 and a simulation version on CloudSim (Calheiros et al., 2011) for repeatable and fast validation. We conducted both simulations and real experiments to demonstrate the efficacy of our approach by comparing the results with the benchmark approaches.

In the future, we plan to further optimize our system by incorporating the following features:

- selection of spot groups according to predicted spot prices in near future;
- dynamic decision of fault-tolerant level and proportion of on-demand instances according to volatility of the spot market using machine leaning technologies;
- an interface that allows web application providers to plug in different workload prediction techniques into the auto-scaling system to achieve proactive auto-scaling; and
- utilization of spot groups across different availability zones.

Acknowledgment

We thank the anonymous reviewers, Adel Nadjaran Toosi, Xunyun Liu, Yaser Mansouri, Bowen Zhou, and Nikolay Grozev for their valuable comments in improving the quality of the paper. This work is supported by Australian Research Council Future Fellowship.

References

Amazon, Amazon ec2 spot instances. URL ⟨http://aws.amazon.com/ec2/spot-instances/⟩.

Amazon, Amazon spot fleet api. URL ⟨https://aws.amazon.com/blogs/aws/new-resource-oriented-bidding-for-ec2-spot-instances/⟩.

Amazon, Ec2 spot instance termination notices. URL ⟨https://aws.amazon.com/blogs/aws/new-ec2-spot-instance-termination-notices/⟩.

Amazon, 2015. Auto scaling, URL ⟨http://aws.amazon.com/autoscaling/⟩.

Barrett E, Howley E, Duggan J. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. Concurr. Comput.: Pract. Exp. 2013;25(12):1656–74.

Binnig, C., Salama, A., Zamanian, E., El-Hindi, M., Feil, S., Ziegler, T., 2015. Spotgres – parallel data analytics on spot instances. In: Proceedings of 2015 31st IEEE International Conference on Data Engineering Workshops (ICDEW), pp. 14–21.

Calheiros RN, Ranjan R, Beloglazov A, De Rose CA, Buyya R. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw.: Pract. Exp. 2011;41(1):23–50.

Caron E, Desprez F, Muresan A. Pattern matching based forecast of non-periodic repetitive behavior for cloud clients. J. Grid Comput. 2011;9(1):49–64.

Changbing, C., Bu Sung, L., Xueyan, T., 2014. Improving hadoop monetary efficiency in the cloud using spot instances. In: Proceedings of 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 312–319.

Chohan, N., Castillo, C., Spreitzer, M., Steinder, M., Tantawi, A., Krintz, C., See spot run: using spot instances for mapreduce workflows. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, USENIX Association, pp. 7.

Costache, S., Parlavantzas, N., Morin, C., Kortas, S., 2012. Themis: economy-based automatic resource scaling for cloud systems. In: Proceedings of 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp. 367–374.

Dean J, Luiz BA. The tail at scale. Commun. ACM 2013;56(2):74–80.

Dutreilh, X., Kirgizov, S., Melekhova, O., Malenfant, J., Rivierre, N., Truck, I., 2011. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In: Proceedings of the Seventh International Conference on Autonomic and Autonomous Systems (ICAS 2011), pp. 67–74.

Dutta, S., Gera, S., Akshat, V., Viswanathan, B., 2012. Smartscale: automatic application scaling in enterprise clouds. In: Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), pp. 221–228.

Eldin, A.A., Rezaie, A., Mehta, A., Razroev, S., Sjo, X., Stedt-de Luna, S.S., Seleznjev, O., Tordsson, J., Elmroth, E., 2014. How will your workload look like in 6 years? Analyzing wikimedia's workload. In: Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E), pp. 349–354.

Fernandez, H., Pierre, G., Kielmann, T., 2014. Autoscaling web applications in heterogeneous cloud infrastructures. In: Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E), pp. 195–204.

Gandhi, A., Dube, P., Karve, A., Kochut, A., Li, Z., 2014. Modeling the impact of workload on cloud resource scaling. In: Proceedings of 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 310–317.

Gandhi, A., Dube, P., Karve, A., Kochut, A., Zhang, L., 2014. Adaptive, model-driven autoscaling for cloud applications. In: Proceedings of the 11th International Conference on Autonomic Computing (ICAC 14). USENIX Association, Philadelphia, PA, pp. 57–64.

Han, Z., Miao, P., Xinxin, L., Xiaolin, L., Yuguang, F., 2012. Optimal resource rental planning for elastic applications in cloud market. In: Proceedings of 2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), pp. 808–819.

Han R, Ghanem MM, Guo L, Guo Y, Osmond M. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. Fut. Gener. Comput. Syst. 2014;32:82–98.

He, X., Shenoy, P., Sitaraman, R., Irwin, D., 2015. Cutting the cost of hosting online services using cloud spot markets. In: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. ACM, Portland, Oregon, pp. 207–218.

Herbst NR, Huber N, Kounev S, Amrehn E. Self-adaptive workload classification and forecasting for proactive resource provisioning. Concurr. Comput.: Pract. Exp. 2014;26(12):2053–78.

Hsuan-Yi, C., Simmhan, Y., 2014. Cost-efficient and resilient job life-cycle management on hybrid clouds. In: Proceedings of 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS), pp. 327–336.

Islam S, Keung J, Lee K, Liu A. Empirical prediction models for adaptive resource provisioning in the cloud. Fut. Gener. Comput. Syst. 2012;28(1):155–62.

Jangjaimon I, Nian-Feng T. Effective cost reduction for elastic clouds under spot instance pricing through adaptive checkpointing. IEEE Trans. Comput. 2015;64(2):396–409.

Jiang, J., Lu, J., Zhang, G., Long, G., 2013. Optimal cloud resource auto-scaling for web applications. In: Proceedings of 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). IEEE, Delft, Netherlands, pp. 58–65.

Jingqi, Y., Chuanchang, L., Yanlei, S., Zexiang, M., Junliang, C., 2013. Workload predicting-based automatic scaling in service clouds. In: Proceedings of 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD), pp. 810–815.

Jung, D., Chin, S., Chung, K., Yu, H., Gil, J., 2011. An efficient checkpointing scheme using price history of spot instances in cloud computing environment. In: Lecture Notes in Computer Science, vol. 6985. Springer, Berlin, Heidelberg, pp. 185–200 (book section 16).

Lorido-Botran T, Miguel-Alonso J, Lozano J. A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. 2014;12(4):559–92.

Mazzucco, M., Dumas, M., 2011. Achieving performance and availability guarantees with spot instances. In: Proceedings of 2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC), pp. 296–303.

Ming, M., Humphrey, M., 2012. A performance study on the VM startup time in the cloud. In: Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), pp. 423–430.

Poola D, Ramamohanarao K, Buyya R. Fault-tolerant workflow scheduling using spot instances on clouds. Proc. Comput. Sci. 2014;29:523–33.

RightScale, 2015. Understanding the voting process. URL ⟨https://support.rightscale.com/12-Guides/RightScale_101/System_Architecture/RightScale_Alert_System/Alerts_based_on_Voting_Tags/Understanding_the_Voting_Process/⟩.

Roy, N., Dubey, A., Gokhale, A., 2011. Efficient autoscaling in the cloud using predictive models for workload forecasting. In: Proceedings of 2011 IEEE International Conference on Cloud Computing (CLOUD). IEEE, Washington DC, USA, pp. 500–507.

Salah, K., Elbadawi, K., Boutaba, R., 2015. An analytical model for estimating cloud resources of elastic services. J. Netw. Syst. Manag., pp. 1–24.

Sangho Y, Andrzejak A, Kondo D. Monetary cost-aware checkpointing and migration on Amazon cloud spot instances. IEEE Trans. Serv. Comput. 2012;5(4):512–24.

Sharma, P., Lee, S., Guo, T., Irwin, D., Shenoy, P., 2015. Spotcheck: designing a derivative iaas cloud on the spot market. In: Proceedings of the Tenth European Conference on Computer Systems. ACM, Bordeaux, France, pp. 16:1–16:15.

Sifei, L., Xiaorong, L., Long, W., Kasim, H., Palit, H., Hung, T., Legara, E.F.T., Lee, G., 2013. A dynamic hybrid resource provisioning approach for running large-scale computational applications on cloud spot and on-demand instances. In: Proceedings of 2013 International Conference on Parallel and Distributed Systems (ICPADS), pp. 657–662.

Singh R, Sharma P, Irwin D, Shenoy P, Ramakrishnan KK. Here today, gone tomorrow: exploiting transient servers in datacenters. Intern. Comput. IEEE 2014;18(4):22–9.

Srirama, S.N., Ostovar, A., 2014. Optimal resource provisioning for scaling enterprise applications on the cloud. In: Proceedings of 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 262–271.

Subramanya, S., Guo, T., Sharma, P., Irwin, D., Shenoy, P., 2015. Spoton: a batch computing service for the spot market. In: Proceedings of the Sixth ACM Symposium on Cloud Computing. ACM, Big Island, Hawai'i, pp. 329–341.

Upendra, S., Shenoy, P., Sahu, S., Shaikh, A., 2011. A cost-aware elasticity provisioning system for the cloud. In: Proceedings of 2011 31st International Conference on Distributed Computing Systems (ICDCS), pp. 559–570.

Urdaneta G, Pierre G, van Steen M. Wikipedia workload analysis for decentralized hosting. Comput. Netw. 2009;53(11):1830–45.

Urgaonkar B, Shenoy P, Chandra A, Goyal P, Wood T. Agile dynamic provisioning of multi-tier internet applications. ACM Trans. Auton. Adapt. Syst. (TAAS) 2008;3(1):1.

van Baaren E.-J., 2009. Wikibench: A Distributed, Wikipedia based Web Application Benchmark (Master's thesis). VU University Amsterdam.

van Baaren, E.-J., 2015. Wikipedia access trace. URL ⟨http://www.wikibench.eu/?page_id=60⟩.

Voorsluys, W., Buyya, R., 2012. Reliable provisioning of spot instances for compute-intensive applications. In: Proceedings of 2012 IEEE 26th International Conference on Advanced Information Networking and Applications (AINA), pp. 542–549.

Wei, F., ZhiHui, L., Jie, W., ZhenYin, C., 2012. Rpps: a novel resource prediction and provisioning scheme in cloud data center. In: Proceedings of 2012 IEEE Ninth International Conference on Services Computing (SCC), pp. 609–616.

Wilder, B., 2012. Horizontally scaling compute pattern. In: Cloud Architecture Patterns: using Microsoft Azure. O'Reilly Media, Inc.

Xiangping B, Jia R, Cheng-Zhong X. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. IEEE Trans. Parallel Distrib. Syst. 2013;24(4):681–90.

Zafer, M., Yang, S., Kang-Won, L., 2012. Optimal bids for spot vms in a cloud for deadline constrained jobs. In: Proceedings of 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), pp. 75–82.