Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

Near-optimal Allocation of VMs from IaaS Providers by SaaS Providers

Arwa Aldhalaan
aaldhala@masonlive.gmu.edu

Daniel A. Menascé
menasce@gmu.edu

Technical Report GMU-CS-TR-2015-12

Abstract

Software as a Service (SaaS) allows companies and individuals to use software, hosted and managed by a SaaS provider, on a pay-per-use basis instead of paying for the entire upfront, maintenance, and upgrade cost. SaaS providers can lease their computing infrastructure to instantiate VMs that run their software services from Infrastructure as a Service (IaaS) providers on a pay per use basis. SaaS customers can subscribe to and unsubscribe from a software service at anytime. Thus, the SaaS cloud provider should dynamically determine the number of needed VMs to run software services as a function of the demand in a way that minimizes the SaaS cost of using VMs from an IaaS but at the same time guaranteeing an agreed upon Quality of Service (QoS) to the SaaS customers. This paper presents two heuristic techniques that can be used by SaaS providers to determine the type and quantity of VMs to be leased in order to best satisfy customer demands for software services. Our experiments showed that the number of states visited by the proposed method is very low (on the order of 10^{-4} of the entire space) while the solution obtained is 2% more expensive in most cases, 13% more expensive in a few cases, and 31% more expensive in only one case, when compared with the optimal solution.

1 Introduction

Software as a Service (SaaS) [10] allows companies and individuals to use software, hosted and managed by a SaaS provider, on a pay-per-use basis instead of hosting software in their own datacenters and paying for the entire upfront, maintenance, and upgrade cost. SaaS providers can lease their computing infrastructure to instantiate VMs that run their software services from Infrastructure as a Service (IaaS) [10] providers on a pay per use basis. We assume that SaaS customers can subscribe to and unsubscribe from a software service at anytime. Thus, the SaaS cloud provider should dynamically determine the number of needed VMs to run software services as a function of the demand in a way that minimizes the SaaS cost of using VMs from an IaaS but at the same time guaranteeing an agreed upon Quality of Service (QoS) to the SaaS customers.

Providing an individual VM dedicated to each customer to run a software service could lead to substantial waste of resources and high infrastructure costs. Thus, an efficient way for resource utilization and cost reduction is for SaaS providers to employ a multi-tenancy approach [18, 12] where several customers (tenants) can subscribe to the same application that is already running on a specific VM, such that this application behaves for each tenant as if this tenant were the sole user of the application. Consequently, SaaS providers can run the same application for multiple customers in the same computing environment to increase the utilization of resources. At the same time, SaaS providers should maintain response time SLAs and other resource constraints at all times.

This paper solves the problem of determining how SaaS providers can optimally manage the dynamic nature of customer requests in a heterogeneous environment in which VMs are of different capacities, cost, and computing power. SaaS providers need to determine how many and what type of VMs to instantiate in order to satisfy a given demand for software services while meeting response time SLAs.

The main contributions of this paper are (1) a heuristic solution, called `ScaleUpDown`, which is based on hill-climbing and provides a near optimal solution very close to the optimal solution while visiting a very small fraction of the solution space; (2) a simpler heuristic called `FillSlotsFirst` that does not perform as well as `ScaleUpDown` but has a lower computational complexity. The two heuristics were extensively evaluated and compared.

The rest of this report is organized as follows. Sec-

tion 2 presents the notation and assumptions used in the paper. The next section formalizes the optimization problem. Section 4 describes the two heuristics presented in the paper. The next section describes the results of the experiments. Section 6 discusses related work. Finally, section 7 presents concluding remarks.

2 Problem Formalization and Notation

We assume that customers of a SaaS provider can subscribe/unsubscribe to software applications offered by the SaaS provider (see Fig. 1). Each application provided by the SaaS provider is offered at different QoS levels. With each request to the SaaS provider, a customer informs about the application it wants, its desired QoS level, and the number of users to be added or removed from the subscription of that application at that QoS level. For example, customer s may send a request (a, q, u, s) to add u users as subscribers of application a at QoS level q. A positive value for u represents additional subscribers to the application and a negative value a decrease in the number of subscribers.

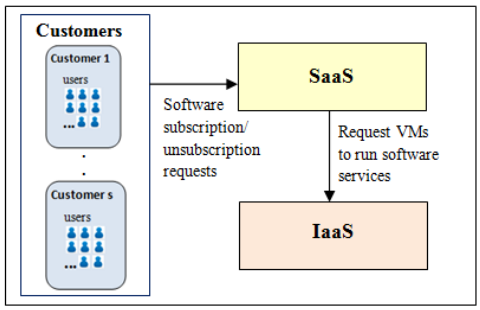

Figure 1: Customers subscribe/unsubscribe to software services at requested QoS for a requested number of users. The SaaS provider determines a near optimal number of VMs to be requested from an IaaS provider to run these software services.

The SaaS provider uses VMs from an IaaS provider to instantiate the software applications it offers to its customers. The IaaS provider offers several types of VMs with different capacities and different cost per unit time. The goal of the SaaS provider is to minimize what it has to pay the IaaS provider while meeting the QoS goals of all its subscribers.

We also assume that a VM is only used to run a single application at a given QoS level. It is assumed that the service demands (see [11]) at the CPU and I/O of an application running at each VM type are known and/or can be easily measured by the SaaS provider. This allow us to use analytic queuing network (QN) performance models to determine the maximum number of concurrent users that running the application at that VM while meeting the desired QoS level.

Some notation is in order.

- (a, q, u, s): a request to the SaaS provider from customer s to add (subtract) u users as subscribers to application a at QoS level q ($q \in \{p, g, s, b\}$), where p = Platinum, g = Gold, s = Silver, and b = Bronze.
- t: type of a VM ($t = 1, \cdots, T$).
- n_t: number of VMs of type t used by application a at QoS q.
- m_t: total number of users allocated to VMs of type t hosting application a at QoS q.
- $m_p(a, q)$: value of $\sum_{t=1}^{T} m_t$ prior to adding (subtracting) the u users in the (a, q, u, s) request.
- c_t: amount (in dollars) the SaaS provider has to pay per time unit (in seconds) to the IaaS provider to use one instance of a VM of type t. We assume that the IaaS provider prorates the charges of VMs in the last fractional hour on a per second basis.
- $C_{(a,q,u,s)}$: cost differential (negative or positive) of changing the subscription level by u users for a (a, q, u, s) request.
- $m_{max}\ (a, q, t)$: maximum number of users supported by a VM of type t running application a at QoS level q with a maximum response time $R^*(a, q)$. We also refer to $m_{max}\ (a, q, t)$ as the maximum number of *slots* available at a VM of type t to run application a at QoS q. The value of $m_{max}\ (a, q, t)$ is easily computed by solving a single-class closed QN model for a VM of type t running application a starting from a concurrency level $n = 0$ to the highest value of n such that $R \leq R^*(a, q)$. See e.g., [11] for the Mean Value Analysis equations needed to solve such closed QN models.
- s_t: slack for type t VMs, i.e., the number of available slots on VMs of type t. According to the algorithms presented here, these slots have to reside on a single VM, otherwise one could obtain a lower cost solution by consolidating slots across several type t VMs and potentially reducing the number of VMs needed.
- $\mathcal{X}_{(a,q,s)} = ((n_1, m_1, s_1), \cdots, (n_t, m_t, s_t), \cdots, (n_T, m_T, s_T))$ subset of states of the SaaS provider for (a, q, s). Note that n_t can be written as $\lceil m_t / m_{max}(a, q, t) \rceil$ and s_t can be written as mod $(m_t, m_{max}(a, q, t))$. Therefore, the state $\mathcal{X}_{(a,q,s)}$ can be expressed more succintly as $(m_1, \cdots, m_t, \cdots, m_T)$.
- $\varphi = \left\{ \mathcal{X}_{(a,q,s)} \ \forall\ a, q, s \right\}$: set of all states of the SaaS provider.

3 Optimization Model

To optimally satisfy each incoming request, we need to determine how many VMs of each type are needed, and the number of users in each VM subject to response time constraints with the goal of minimizing the total cost. To satisfy a request (a, q, u, s) the SaaS provider needs to do a combination of the following actions. If $u > 0$, the actions include adding more users to the already allocated VMs for (a, q) and/or adding more VMs to support the additional number of users. If $u < 0$, the actions include reducing the number of users in the already allocated VMs for (a, q) and/or reducing the number of allocated VMs. In either case, because of the assumption that a VM cannot run different applications or an application at different QoS levels, satisfying a request (a, q, u, s) does not interfere with allocations for different values of a, q, and s.

Therefore, the cost optimization problem faced by a SaaS provider when receiving an (a, q, u, s) request can be expressed as: Given a request (a, q, u, s), the values of $m_{max}(a, q, t)\ \forall\ t \in \{1, \cdots, T\}$, and $m_p(a, q)$.

$$\text{Minimize } C = \sum_{t=1}^{T} c_t \cdot \lceil m_t / m_{max}(a, q, t) \rceil \tag{1}$$

s.t.

$$\sum_{t=1}^{T} m_t = m_p(a, q) + u. \tag{2}$$

$$m_t \in \mathbb{Z}\ \forall\ t \in \{1, \cdots, T\}. \tag{3}$$

The decision variables for this optimization problem are $m_t\ \forall\ t\ \in \{1, \ldots, T\}$, i.e., the total number of users allocated to each VM type.

The QoS level q used throughout the rest of this paper determines the response time of the application $R(a, q)$. In order to compute $m_{max}(a, q, t)$, we need to solve a single-class closed Queuing Network (QN) model [11] for each VM type using the service demands of the application in each of the VM resources as parameters. Because of our assumption that a VM hosts a single application a at a QoS level q, we can use a single-class QN model. In a closed QN, the number of customers in the QN is fixed and represents the concurrency level, which in our case indicates the number of users using application a at a VM at QoS level q. The input parameters of a closed QN model are the number of customers in the QN, and the service demands at each resource, defined as the total average service time provided by a resource to a given class of requests. We denote the processing (i.e., CPU) and I/O service demands of application a at a VM of type t as $D^t_{\text{cpu,a}}$ and $D^t_{\text{I/O,a}}$, respectively. Thus, the maximum number of users in a VM of QoS q running application a can be determined using a single-class queuing network because we assume that a VM cannot run different applications or an application at different QoS levels.

Due to the highly combinatorial and non-linear nature of the problem, we develop a heuristic-based combinatorial search method to find a near-optimal solution to this optimization problem.

4 Heuristic Search

The optimization problem described in the previous section has a feasible state space that grows in a combinatorial way. The number of feasible states is given by

$$\binom{T - 1 + m_p(a, q) + u}{T - 1}. \tag{4}$$

Equation (4) represents the number of ways in which we can allocate the $m_p(a, q) + u$ users to the T types of VMs. Thus, a request for 100 users (i.e, $u = 100$) that finds 4,000 users (i.e., $m_p(a, q) = 4,000$) allocated to four VM types (i.e., $T = 4$) results in approximately 11.5×10^9 states. Therefore, an efficient heuristic is required to find a near-optimal solution to this optimization problem, which must be solved by the SaaS in almost real-time. Our heuristic solution, called `ScaleUpDown`, uses the hill climbing search technique [14] to determine how many VMs of each type are needed, and how many users should be allocated to each VM type.

4.1 The ScaleUpDown Algorithm

A high-level description of the `ScaleUpDown` algorithm (see Algorithm 1) is as follows. If the request is for removing users, it removes users from each of the VM types and releases VMs that became empty back to the IaaS. If the request is for adding users it tries to add all new users to empty slots in already allocated VMs. If more slots are needed, the algorithm obtains these slots from new VMs of type 1 (the cheapest) and then uses a hill-climbing method to re-balance the allocation to achieve a near-optimal solution. Note that this algorithm is different from existing auto-scaling methods offered by IaaS providers such as Amazon EC2. IaaS providers are able to measure the utilization and availability of a user's VMs. However, these cloud providers are not able to monitor application response time in order to respond to that. This can only be done by a SaaS provider because it has visibility of the application.

A more detailed description of the `ScaleUpDown` algorithm is the following. It receives a request (a, q, u, s) and starts by invoking the `AddRemoveUsers` function described in Algorithm 2. This algorithm determines if the request is for removing users (i.e., $u < 0$) or adding users (i.e., $u > 0$). If it is a removal request, `AddRemoveUsers` removes u users starting from the smallest capacity VMs to the largest capacity ones (lines 4-15). This is done in order to expedite the release of more VMs. The removal of users for a given VM type is done with the `RemoveUsers` algorithm invoked in lines 8 and 13 of the

Algorithm 1 ScaleUpDown Algorithm

1: `ScaleUpDown` $(a, q, s, m, u, T, \varphi)$
2: /* If $u < 0$ remove users starting from partially filled VMs. If $u > 0$ add as many users as possible to partially filled slots */
3: AddRemoveUsers (a, q, s, m, u, T, φ);
4: **if** $u > 0$ **then**
5: /* Allocate remaining users in new VMs of type 1 */
6: ExtraVMs $\leftarrow \lceil u / m_{max}(a, q, 1) \rceil$;
7: $n_1 \leftarrow n_1 +$ ExtraVMs;
8: AddUsers $(a, q, u, 1)$; /* See Algorithm 4 */
9: $\mathcal{X}_{(a,q,s)} \leftarrow$ SaaSHill ($\mathcal{X}_{(a,q,s)}$, T, MaxRestarts, MaxIterations) ;
10: **end if**
11: **Return** (φ)

`AddRemoveUsers` algorithm. If the request is for user allocation, the `AddRemoveUsers` algorithm loops over all VM types from the least costly to the more costly and fills available slots in these VM types using the `AddUsers` algorithm (line 20). This algorithm receives as input the number u of users to be added to type t VMs and returns in u the remaining number of users yet to be allocated. If all users can be allocated into available slots, the algorithm breaks from the loop (line 22) and returns u (line 25). Note that allocating users to empty slots does not increase the cost to the SaaS provider because these VMs are already being paid to the IaaS.

After executing the `AddRemoveUsers` algorithm, the `ScaleUpDown` algorithm checks if there are still users to be allocated (i.e., $u > 0$). If that is the case (lines 6-9), the algorithm computes the number of extra VMs of type 1 (the cheapest) needed for the remaining users (lines 7-8). The `AddUsers` algorithm is invoked to allocate these users to these additional VMs (line 8). Finally, the cloud is rebalanced using a hill-climbing algorithm, `SaaSHill` (line 9), to be described later.

The `RemoveUsers` algorithm (see Algorithm 3) checks if all VMs of type t are full ($s_t = 0$). In that case (lines 5-7), the number of users u to be removed is used to determine the number of VMs to be returned back to the IaaS provider and to update n_t (line 6). The slack s_t is also updated (line 7). If there is a type t VM with available slots (i.e., $s_t > 0$) (lines 9-16), n_t and s_t are updated taking into account the number of users, $m_{max}(a, q, t) - s_t$, in the partially full VM.

Algorithm `AddUsers` (Algorithm 4) adds users to available slots in type t VMs if there are any and returns the number of users that remain to be added after all available slots have been filled (line 18). If the users to be added fit into the available slots (i.e., $u \leq s_t$), then the number of available slots is decremented by the number of users to be added (line 7), the number of users added (ua) is made equal to u (line 8), and the number of remaining users to add is set to zero (line 9). Otherwise,

Algorithm 2 AddRemoveUsers Algorithm

1: `AddRemoveUsers` (a, q, s, u, T, φ)
2: **if** $u < 0$ **then**
3: /* remove u users */
4: $u \leftarrow -u$;
5: **for all** t= 1 to T **do**
6: /* remove u users from VMs of type t */
7: **if** $s_t \geq u$ **then**
8: /* remove a total of u users from type t VMs */
9: RemoveUsers (a, q, u, t); /* See Algorithm 3 */
10: Break
11: **else**
12: /* remove all users allocated to VMs of type t */
13: $u = u - s_t$;
14: RemoveUsers (a, q, u, t); /* See Algorithm 3 */
15: **end if**
16: **end for**
17: **Return** (u, φ)
18: **end if**
19: /* Need to add users into empty slots */
20: **for all** t = 1 to T **do**
21: `AddUsers` (a, q, u, t); /* See Algorithm 4 */
22: **if** $u = 0$ **then**
23: Break;
24: **end if**
25: **end for**
26: **Return** (u, φ)

Algorithm 3 RemoveUsers Algorithm

1: `RemoveUsers` (a, q, u, t)
2: /* remove u users from VMs running application a at QoS level q */
3: $m_t \leftarrow m_t - u$;
4: **if** $s_t = 0$ **then**
5: /* all VMs of type t are totally full */
6: $n_t \leftarrow n_t - \lfloor u / m_{max}(a, q, t) \rfloor$; /* deallocate these VMs */
7: $s_t \leftarrow mod\ (u, m_{max}(a, q, t))$
8: **else**
9: FSLots $\leftarrow m_{max}(a, q, t) - s_t$ /* no. filled slots */
10: **if** $u <$ FSlots **then**
11: $s_t \leftarrow s_t + u$
12: **else**
13: /* deallocate these VMs */
14: $n_t \leftarrow n_t - (1 + \lfloor (u - \text{FSlots}) / m_{max}(a, q, t) \rfloor)$;
15: $s_t \leftarrow mod\ (u - \text{FSlots}, m_{max}(a, q, t))$
16: **end if**
17: **end if**

Algorithm 4 AddUsers Algorithm

```
1: AddUsers (a, q, u, t)
2: /* add users to available slots of VM of type t that
   host applications of the type (a, q). */
3: if s_t > 0 then
4:   /* there is a partially filled VM */
5:   if u ≤ s_t then
6:     /* all users can be added to the partially filled
       VM */
7:     s_t ← s_t − u;
8:     ua ← u; /* users added */
9:     u ← 0;
10:  else
11:    /* not all users fit in the partially filled VM */
12:    u ← u − s_t;
13:    ua ← s_t; /* users added */
14:    s_t ← 0;
15:  end if
16: end if
17: m_t ← m_t + ua;
18: Return (u); /* return remaining number of users to
    be added */
```

Algorithm 5 Hill Climbing

```
1: SaaSHill (X_(a,q,s), T, MaxRestarts, MaxIterations)
2: for all NumRestarts= 1 to MaxRestarts do
3:   for all NumIterations = 1 to MaxIterations do
4:     /* Perform local search */
5:     X*_(a,q,s) ← Neighborhood (X_(a,q,s), T) ; /* Re-
       turns the state with lowest cost in the neigh-
       borhood */
6:     if Cost(X*_(a,q,s)) = Cost(X_(a,q,s)) then
7:       Break; /* No Improvement */
8:     else
9:       X_(a,q,s) ← X*_(a,q,s); /* Cost is reduced */
10:    end if
11:  end for
12:  LocalOpt[NumRestarts] ← X_(a,q,s);
13:  /*New Restart*/
14:  X_(a,q,s) ← NewState (X_(a,q,s));
15: end for
16: Return argmin_LocalOpt[i] {Cost(LocalOpt[i])}
```

not all users fit into available slots (lines 11-14). The number of users tha remain to be added is computed as the number of users intended to be added minus the number of available slots (line 12). The number of added users is equal to the number of available slots (line 13) and all available slots become full (line 14).

The `SaasHill` algorithm (Algorithm 5) starts from a current state $\mathcal{X}_{(a,q,s)}$ and iteratively explores a fraction of the solution space by creating neighborhoods of states using the `Neighborhood` function described in Algorithm 6. The `SaasHill` algorithm performs `MaxRestarts` local searches (line 2) to reduce the possibility that the global search be stuck in a local optimum, a well-known drawback of this type of search. The first search starts from $\mathcal{X}_{(a,q,s)}$ and iterates over `NumIterations` neighborhoods (line 3). The other searches start from a feasible state (i.e., that satisfies the response time constraint) obtained by the function `NewState` (line 14) not shown in detail here.

Each local search (lines 3-11) finds a neighborhood of the state $\mathcal{X}_{(a,q,s)}$ and returns the state $\mathcal{X}^*_{(a,q,s)}$ in the neighborhood with the minimum cost (line 5). If there is no cost improvement, the local search terminates and restarts from a new state (line 14). Once the local search ends (line 11), another is restarted until the limit of restarts, `MaxRestarts`, is reached. After all local searches are executed, the algorithm returns the state with the minimum cost among those identified by all local searches (line 16).

The `Neighborhood` algorithm (see Algorithm 6) builds a neighborhood of a state $\mathcal{X}_{(a,q,s)}$ by adding states according to two methods: (1) For each VM of type t (t from 2 to T) add a new VM of this type and move as many users as possible from lower capacity VMs to the new VM of type t (lines 4-13). In the process, VMs of lower capacity that become empty are released back to the IaaS provider. The function `Increase` (invoked in line 6 and implemented by Algorithm 7) performs this step. (2) For each VM of type t (t from 2 to T) remove a VM of this type and move its users to lower capacity VMs (lines 14-24). In the process, it may be necessary to add VMs of lower capacity. The function `Decrease` (invoked in line 17 and implemented by Algorithm 8) performs this step. The functions `Increase` and `Decrease` return a new state. If the cost of that new state is less than the minimum cost obtained so far, the returned state becomes the state with the minimum cost and its cost becomes the current minimum cost (lines 7-11 and 18-22 of the `Neighborhood` algorithm). Finally, the `Neighborhood` function returns a state with the minimum cost in the neighborhood.

The `Increase` and `Decrease` methods use the `AddUsers` and `RemoveUsers` functions described previously.

4.2 `ScaleUpDown` Algorithm Example

The following example shows different states visited by the hill climbing search to find a near optimal solution. However, none of the states visited, except for the near-optimal solution found by the hill climbing method, result in actual reallocation of VMs. Consider the case of an application a with platinum ($q = p$) QoS. Suppose there are three VM types ($T = 3$) and that each VM type has the following maximum number of users to not violate the response time constraint: $m_{max}(a, p, 1) = 10$, $m_{max}(a, p, 2) = 12$, and $m_{max}(a, p, 3) = 25$ as shown in Table 1. Suppose that the cost of operating each VM type is as follows: $c_1 = \$5$ per hour, $c_2 = \$6.5$ per hour, and $c_3 = \$9$ per hour. The fourth row of the table shows the

Table 1: Allocation Example.

	$c_1 = 5$				$c_2 = 6.5$			$c_3 = 9$		
	$m_{max}(1,p,1) = 10$				$m_{max}(1,p,2) = 12$			$m_{max}(1,p,3) = 25$		
	Type 1 VMs				Type 2 VMs			Type 3 VMs		
Current State	10		7		12		8	19		
Request to add 30 users										
Filling the slots	10		10		12		12	25		
State 1: \$42, add VMs t=1	10	10	10	7	12		12	25		
Increase phase										
State 2: \$43, add VMs t=2	10	10	5		12	12	12	25		
State 3: \$41, add VMs t=3	10		2		12		12	25	25	
State 4: \$37, add VMs t=3	10		2					25	25	24
Decrease phase										
State 5: \$37, remove VMs t=2	10		2					25	25	24
State 6: \$43, remove VMs t=3	10	10	10	5	2			25		24
State 7: \$47.5, remove VMs t=3	10		2		12	12	1	25		24

current allocation of two VMs of type 1 with 10 users in the first VM and 7 users in the second, two VMs of type 2 with 12 users in the first VM and 8 users in the second, and one VM of type 3 with 19 users.

Each row of Table 1 shows the states tested by the `ScaleUpDown` algorithm. At each state, the cost is calculated and the state with the minimum cost is selected. Note that no changes are committed to the cloud environment until this search is over and a better solution is found. When a request arrives for 30 new users, the sixth row shows the occupation of the 13 empty slots across all VMs of all types. State 1 in the table shows the state resulting by allocating the remaining 17 users to the smallest type VM, which is of type 1. The cost of state 1 is \$42. Then, the `Increase` process starts by adding a VM of type 2 and moving 12 users from VMs of type 1 as shown in state 2 resulting in a cost of \$43. State 3 adds a VM of type 3 and moves 25 users from VMs of type 1 based on state 1 since it has the lowest cost so far. The cost of state 3 is \$41, which is the current minimum cost, so it is going to be the base state for further states. State 4 adds a VM of type 3 and moves 24 users from VMs of type 2 based on state 3. The cost of state 4 is \$37. Thus, state 4 has the minimum current cost and therefore will be the base state for further states. Now, the `Decrease` process starts with state 5 by removing users from a VM of type 2 and moving them into VMs of type 1. But since there are no VMs of type 2, no change happens in this state. State 6 removes one VM of type 3 and moves 25 users into VMs of type 1 based on state 4. The cost of state 6 is \$43. State 7 removes a VM of type 3 and moves 25 users from a VM of type 3 to VMs of type 2 based on state 4. The cost of state 7 is \$47.5. Up to this point, all states have been tested and only the state with the lowest cost is returned as the lowest cost solution for this neighborhood, which is state 4.

4.3 The `FillSlotsFirst` Algorithm

We also designed an experimented with another heuristic called `FillSlotsFirst` (see Algorithm 9), which is much simpler than the `ScaleUpDown` method described above. The `FillSlotsFirst` algorithm starts by adding users to available slots for the same (a, q) pairs using the same `AddRemoveUsers` as the `ScaleUpDown` algorithm (line 5). Then, if there are still remaining users to be allocated, the method allocates them to the VM type that provides the least cost (lines 9-10). The number of available slots in the VM type chosen to allocate the remaining users is computed in lines 11-15. One of the main differences between the `FillSlotsFirst` and `ScaleUpDown` methods is that the former does not use a hill-climbing method to rebalance the cloud. Therefore, `FillSlotsFirst` is simpler to implement and has a low computational complexity of $\mathcal{O}(T)$. On the other hand, the computational complexity of the `ScaleUpDown` algorithm is $\mathcal{O}$ (MaxRestarts $\times$ MaxIterations $\times T^2$).

4.4 Comparison With the Optimal Solution

To compare our algorithm with the optimal solution, we perform an exhaustive search of the entire solution space for a small problem. The implementation shown in Algorithm 10 works for three VM types. It is easy to see how it can be extended to more VM types. The algorithm tests all different combinations of the total number of users allocated to each VM type (lines 3-9). Then, the algorithm calculates the number of VMs used of each type for a specific allocation (lines 10- 13). After that, the algorithm calculates the cost of each allocation (lines 15 -18) and returns the allocation with the minimum cost (line 21). Clearly, as we indicated before, the optimal algorithm is only practical for a very small problem sizes.

Algorithm 6 Neighborhood

```
Neighborhood (X_(a,q,s), T )
MinCost ← Cost(X_(a,q,s));
/* Increase the number of VMs */
for all t=2 to T do
    for all v=1 to t-1 do
        X^Temp_(a,q,s) ← Increase ( X_(a,q,s), t, v);
        if Cost(X^Temp_(a,q,s)) < MinCost then
            /* Change to the allocation with lower cost */
            MinCost ← Cost(X^Temp_(a,q,s));
            X_(a,q,s) ← X^Temp_(a,q,s) ;
        end if
    end for
end for
/* Decrease the number of VMs */
for all t=2 to T do
    for all v=1 to t-1 do
        X^Temp_(a,q,s) ← Decrease (X_(a,q,s), t, v)
        if Cost(X^Temp_(a,q,s)) < MinCost then
            /* Change to the allocation with lower cost */
            MinCost ← Cost(X^Temp_(a,q,s));
            X_(a,q,s) ← X^Temp_(a,q,s) ;
        end if
    end for
end for
Return (X_(a,q,s))
```

Algorithm 7 Increase

```
Increase (X_(a,q,s), t, j)
/* Add a new VM of type t and move users from VMs of type j to the new VM of type t (t > j). Remove the unused VMs of type j.
n_t ← n_t + 1;
u ← AddUsers (a, q, m_max(a, q, t), t);
RemoveUsers (a, q, m_max(a, q, t), j);
Return (X_(a,q,s))
```

Algorithm 8 Decrease

```
Decrease (X_(a,q,s), t, j)
/* Remove a VM of type t and move its users to VMs of type j (t > j). */
NumUsers ← max_j(X_t[j]);
RemoveUsers (a, q, NumUsers, t);
u ← AddUsers (a, q, NumUsers, j);
Return (X_(a,q,s))
```

Algorithm 9 FillSlotsFirst

```
FillSlotsFirst ( a, q, u, T)
/* Finds the optimal allocation for (a, q, u, s)*/
/* If u < 0 remove users starting from partially filled VMs. */
/* If u > 0 add as many users as possible to partially filled slots */
AddRemoveUsers (a, q, s, m, u, T, φ);
/* u contains the number of remanining users to be added after filling available slots. */
if u > 0 then
    /* at this point all s_t are zero for all VM types */
    t_min ← argmin_{t=1}^{T} { ⌈u/m_max(a, q, t)⌉ × c_t }
    n_{t_min} ← n_{t_min} + ⌈u/m_max(a, q, t_min)⌉
    if u < m_max(a, q, t_min) then
        s_{t_min} ← m_max(a, q, t_min) − u
    else
        s_{t_min} ← m_max(a, q, t_min)
                  − mod (u, m_max(a, q, t_min))
    end if
end if
Return (φ)
```

5 Experimental Results

Our experiments consider a Poisson arrival stream of user allocation or deallocation requests for software services at a rate of 2 req/sec. Every time a request arrives, a new near-optimal state is computed using the `ScaleUpDown` and `FillSlotsFirst` algorithms described in the previous section.

Table 2 shows the parameters used in the experiments. There are 2 different applications, 3 VM types, 2 different customers. The maximum response time SLA, R^*, given in the table is the same for both applications and is different for each of the four QoS levels. The value of `MaxUsers` is the maximum number of users of a customer that will use each application at a given QoS level. In the experiments, according to the table, we assume this value to be the same for the two applications. So, for example, each customer can have up to 1,150 employees using any of the two applications of platinum QoS level. `TargetUsers` is a value such that once reached, customer requests can be for either increase or decrease of users. For the experiments we consider `TargetUsers` to be 90% of `MaxUsers`. The cost c_t given in the table is typical of IaaS providers such as Amazon's EC2. The values of `MaxIterations` and `MaxRestarts` determine the behavior of the hill-climbing aspect of the `ScaleUpDown` method. The CPU and service demands used in the experiments for each VM type are given in Table 3.

We randomly generated 30 workloads composed of 600 requests each for a total of 18,000 requests by simulating a real cloud environment using Matlab. In each workload, the number of users u for each request was randomly generated between 1 and 70 from a random customer s. The applications are chosen between applications 1 and 2 with equal probability. Initially, all requests are allocation requests (i.e., $u > 0$). At each request gen-

Algorithm 10 Optimal algorithm: tests all possible states and returns the state with minimum cost

```
1: OptimalAlg (a, q, s, u, X_(a,q,s))
2: MinCost ← ∞;
3: for all j=0 to u do
4:   for all y=0 to (u − j) do
5:     i = u − (j + y)
6:     /* Use different combinations of the number of
       users in each VM type */
7:     m_1 ← j;
8:     m_2 ← y;
9:     m_3 ← i;
10:    /* Update the number of VMs used */
11:    n_1 ← ⌈m_1/m_max(a, q, 1)⌉;
12:    n_2 ← ⌈m_2/m_max(a, q, 2)⌉;
13:    n_3 ← ⌈m_3/m_max(a, q, 3)⌉;
14:    /* Compute the total cost and test if it is the
       minimum */
15:    if Cost(X^Temp_(a,q,s)) < MinCost then
16:      MinCost ← Cost(X^Temp_(a,q,s));
17:      X_(a,q,s) ← X^Temp_(a,q,s) ;
18:    end if
19:  end for
20: end for
21: Return (X_(a,q,s), MinCost)
```

eration, the number of allocated users for each (a, q, s) is calculated. If this number exceeds `TargetUsers`, the request becomes an allocation (i.e., $u > 0$) with a 50% chance or a deallocation request (i.e., $u < 0$).

We used the same 30 workloads to compare the `ScaleUpDown` and the `FillSlotsFirst` strategies and computed the average per-request cost and the cumulative cost averaged for all 30 workloads and all requests. All graphs described in what follows show several metrics as a function of time in seconds.

The three curves of Fig. 2 represent average values over all 30 workloads and all requests versus time. The top curve shows the value of `MaxUsers`, the middle one `TargetUsers`, and the bottom curve shows the average number of users. As shown, the bottom curve starts

Table 2: Parameter values used in the experiments.

Parameter	Value
A	2
T	3
S	2
R^*	(97,77,67,57) $\forall\ a, \forall q \in \{p, g, s, b\}$
MaxUsers	(1150,1170,1190,1200) $\forall q \in \{p, g, s, b\}$
TargetUsers	0.90 × MaxUsers
c_1, c_2, c_3	0.036, 0.070, 0.226 $/hour
`MaxIterations`	20
`MaxRestarts`	2

Table 3: CPU and I/O service demands (in sec) for each VM type.

	VM type 1	VM type 2	VM type 3
CPU	0.044	0.019	0.001
I/O	0.019	0.002	0.0102

with allocation requests until the total number of users reaches the value of `TargetUsers`. After that, there is an equal probability of generating deallocation or allocation requests.

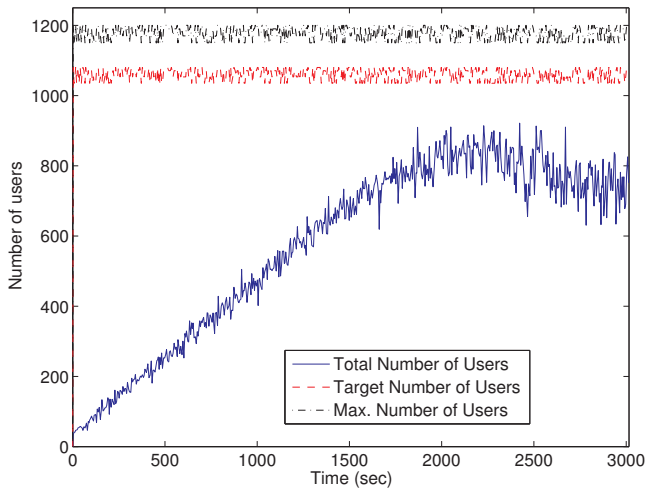

Figure 2: Average values over all 30 workloads and all requests. Top: `MaxUsers`, Middle: `TargetUsers`, and Bottom: Number of users.

Figure 3 compares the current cost (in 10^{-3} $/sec) to the SaaS provider when using `ScaleUpDown` versus `FillSlotsFirst` to lease all needed VMs including the ones required to satisfy the arriving request. The graph shows a slight decrease in cost after about 1,900 sec because by that time some requests are for user deallocations as can be seen in Fig. 2. As shown, there is a clear difference between the two curves. The maximum cost/sec observed for `ScaleUpDown` during the experiment is approximately 2.1×10^{-3} $/sec or close to $5,500/month. The corresponding value for `FillSlotsFirst` is $7,800/month, or 40% higher. The reason is that `FillSlotsFirst` does not try to re-balance the allocation across VMs after slots are filled.

Figure 4 compares the accumulated cost of leasing all VMs including the VMs needed to satisfy the arriving request for `ScaleUpDown` and `FillSlotsFirst`. The separation between the two algorithms is obvious. The accumulated cost for `FillSlotsFirst` increases at a rate of 0.22 ¢/ sec at the end of the experiment while the accumulated cost for `ScaleUpDown` grows at the smaller rate of 0.18 ¢/ sec during the same interval. At time 3,000, the accumulated cost of `FillSlotsFirst` is close to $5,300/month, 30% higher than that of `ScaleUpDown`.

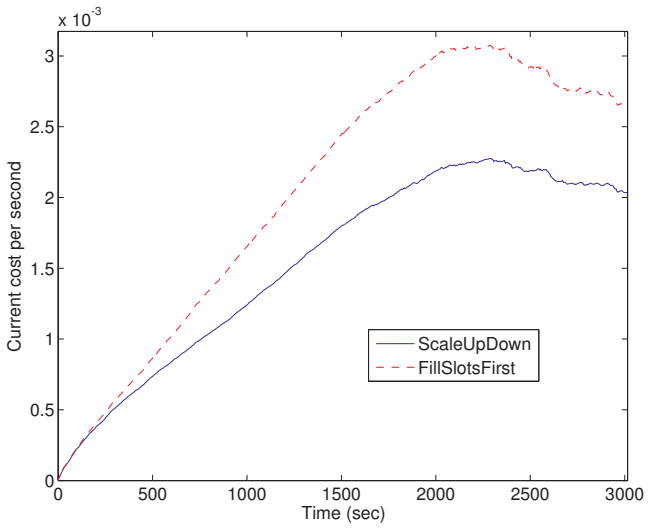

Figure 3: Current cost (in 10^{-3} \$/sec) vs. time for `ScaleUpDown` (bottom) and `FillSlotsFirst` (top).

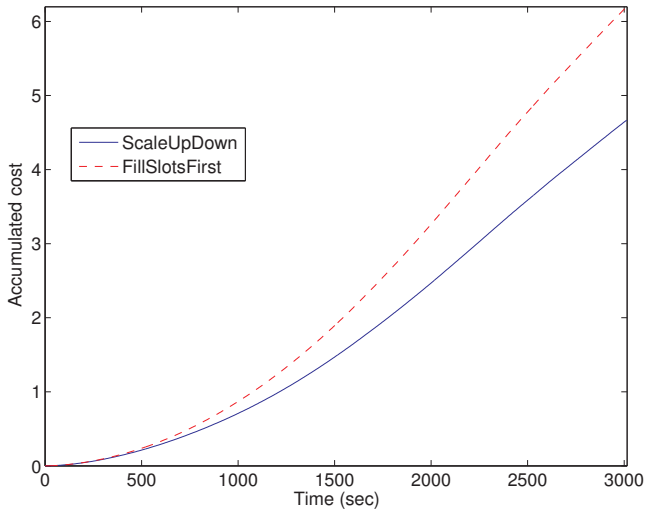

Figure 4: Accumulated cost (in \$) for `ScaleUpDown` (bottom) and `FillSlotsFirst` (top)

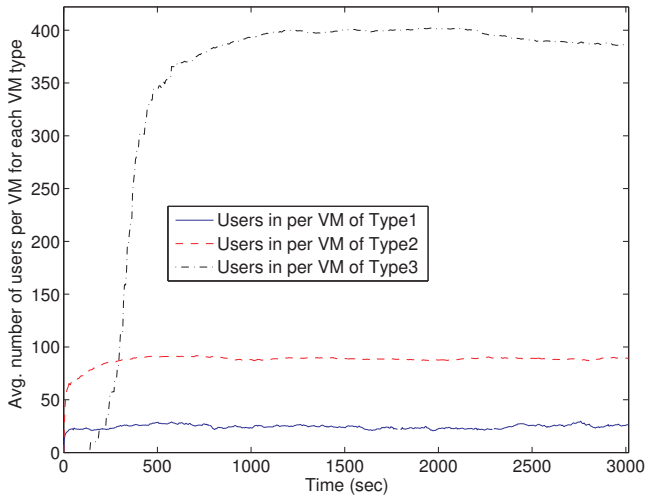

Figure 5: `ScaleUpDown`: Average number of users per VM in each VM type. At time 3000: Type 3 (top), Type 2 (middle), Type 1 (bottom).

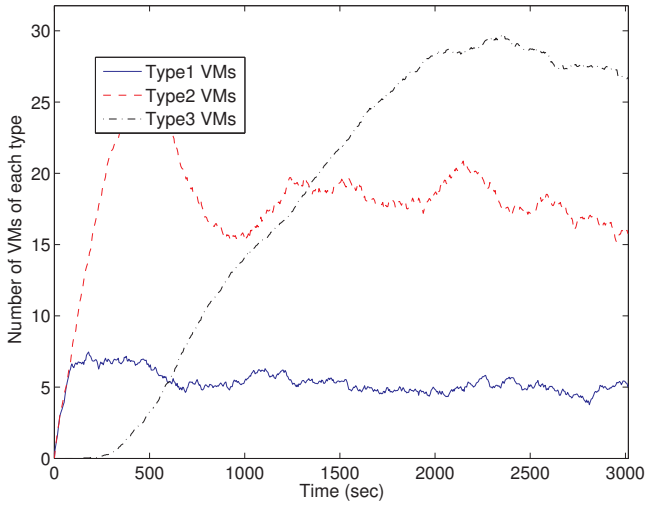

Figure 6: `ScaleUpDown`: Average number of used VMs of each type. At time 3000: Type 3 (top), Type 2 (middle), Type 1 (bottom)

Figure 5 shows the average number of users per VM in each VM type for `ScaleUpDown`. The figure shows that after about 1,000 sec, the average number of users per VM type stabilizes at around 400 for type 3 VMs, 90 for type 2 VMs and 25 for type 1 VMs. The average number of users per VM for all VM types is then approximately 515 users after 1,000 sec have elapsed. Table 2 shows the maximum number of users, R^*, for both applications and all QoS levels. Since all four QoS levels are equally likely and the two applications are also equally likely to be requested, the total maximum number of users per VM for all VM types if all slots were filled would be $2 \times (97 + 77 + 67 + 57) = 596$. So, 515 users represents an average utilization of 86% for all VM types. It is also interesting to observe that type 3 VMs account for 78% (=400/515) of users per VM, type 2 account for 17%, and type 1 for the remaining 5%.

The number of VMs of each type used by `ScaleUpDown` is shown in Fig. 6. Figure 7, the equivalent of Fig. 5 for `FillSlotsFirst`, shows the average number of users per VM in each VM type using `FillSlotsFirst`. As we can see, `FillSlotsFirst`, at least for the parameters used in the experiments, avoids the most expensive type 3 VMs because it prefers VM types that provide the lowest cost per user at all times.

Figure 8 shows the total number of users in each VM type for `ScaleUpDown`. As the number of requests increases over time, more users are allocated to type 3 VMs and less users are allocated to type 1 VMs. The

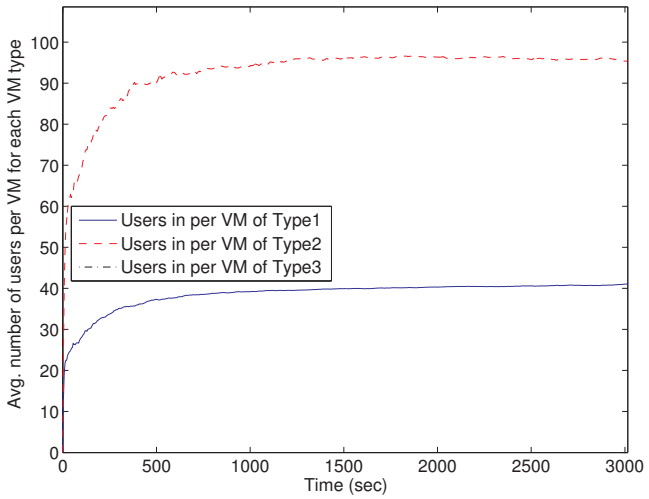

Figure 7: `FillSlotsFirst`: Average number of users per VM in each VM type: Type 2 (top) and Type 1 (bottom).

reason is that type 3 VMs can accommodate more users at a lower cost than multiple smaller size VMs of type 1.

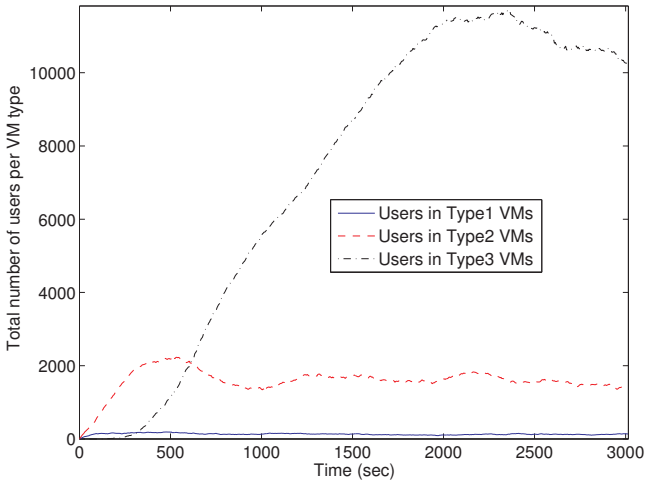

Figure 8: `ScaleUpDown`: Average total number of users in each type of VM. At time 3000: Type 3 (top), Type 2 (middle), Type 1 (bottom).

We also compared the results obtained from `ScaleUpDown` with the optimal results otained by running the `Optimal` algorithm (see Algorithm 10). The results are summarized in Table 4. The first column shows the request. The second column shows the number of users in each VM type as (Max/ScaleUpDown/Opt) which stands for the maximum number of users in each VM type, the number of users allocated using `ScaleUpDown`, and the number of users allocated using the optimal solution. The third column is the cost of the `ScaleUpDown` (SUD) solution. The fourth column is the cost of the optimal solution. The fifth column is the SUD cost divided by the cost of the optimal solution. The sixth column shows the fraction of the total number of states visited by `ScaleUpDwown`. As shown, the number of states visited by `ScaleUpDwown` is very low (on the order of 10^{-4} of the entire state space) while the solution obtained is 2% more expensive in many cases, 13% more expensive in others, and 31% more expensive in only one case.

6 Related Work

There has been significant work on resource allocation in IaaS cloud providers but relatively little work on the resource allocation problem faced by SaaS cloud providers. Some examples of resource allocation work for IaaS providers include [1, 2, 4, 13, 7, 3, 9, 16]. In [6], the reader can find a survey of recent cloud resource management techniques. Frameworks for minimizing the cost of resources were presented in [18, 5, 21, 22, 15, 17, 16]. In [18], resource allocation algorithms were presented for SaaS providers to minimize infrastructure cost and SLA violations such as in response time. Their algorithms are based on mapping and scheduling mechanisms and policies for translating the customers QoS requirements to infrastructure level parameters and allocating VMs to serve their requests. However, they do not use heuristic algorithms. In [21], the authors present evolutionary algorithms to minimize resource usage for SaaS providers and improve execution time. The goal of [22] is to minimize the resources used by the SaaS by clustering its components without violating specific constraints. Their algorithm considers the SaaS resource and communication requirements. The authors in [15] and [17] minimized the infrastructure cost by using a multi-tenant SaaS model where a single instance of a software application serves multiple customers (tenants). But they did not use heuristics for optimization. In [20], the authors present an optimal algorithm that minimizes energy consumption in the context of MapReduce applications. The work in [23] uses simulation to address the problem of minimizing an infrastructure for running a MapReduce job given a completion time target for the job. A cost minimization solution is provided in [19], which presents the design, implementation, and evaluation of a resource management system for cloud computing services used to allocate data center resources dynamically based on application demands and optimizing the number of servers in use. In their algorithms, they use the skewness metric to combine VMs with different resource characteristics to physical resources. The authors in [8] provide capacity planning heuristics that use a utility model for a SaaS. Their utility model considers business aspects related to offering a SaaS application with a goal of increasing the profit.

None of the above mentioned solutions use heuristic search techniques for minimizing the cost of SaaS cloud providers with response time SLAs constraints.

Table 4: Comparison of the `ScaleUpDown` algorithm with the optimal solution

(a,q,u)	(VM1, VM2, VM3)	SUD	Opt	SUD/Opt	Visited
(1, 1, 420)	(41/0/8, 100/99/0, 412/321/412)	8.28E-05	7.28E-05	1.14	2.14E-04
(1, 2, 100)	(41/100/1, 100/0/99, 412/0/0)	3.00E-05	2.94E-05	1.02	1.55E-03
(1, 3, 105)	(41/105/6, 99/0/99, 411/0/0)	3.00E-05	2.94E-05	1.02	1.41E-03
(1, 4, 122)	(40/122/23, 99/0/99, 410/0/0)	3.00E-05	2.94E-05	1.02	7.99E-04
(2, 1, 450)	(41/0/38, 100/99/0, 412/351/412)	8.22E-05	7.28E-05	1.13	1.86E-04
(2, 2, 412)	(41/0/0, 100/99/0, 412/313/412)	8.22E-05	6.28E-05	1.31	2.22E-04
(2, 3, 430)	(41/0/19, 99/99/0, 411/331/411)	8.22E-05	7.28E-05	1.13	2.04E-04
(2, 4, 451)	(41/0/41, 99/99/0, 410/352/410)	8.22E-05	7.28E-05	1.13	1.85E-04
(1, 1, 115)	(41/115/16, 100/0/99, 412/0/0)	3.00E-05	2.94E-05	1.02	1.18E-03
(1, 2, 441)	(41/0/30, 100/99/342, 412/0/411)	8.22E-05	7.28E-05	1.13	1.94E-04

7 Concluding Remarks

SaaS cloud providers dynamically scale the number of needed VMs to run software services depending on the demand. They need to optimally manage the dynamic nature of customer requests in a heterogeneous environment in which VMs are of different capacities, cost, and computing power. Therefore, due to the highly combinatorial and non-linear nature of the problem, we developed a heuristic-based combinatorial search method to find a near-optimal solution to this optimization problem subject to response time constraints with the goal of minimizing the total cost.

Our heuristic solution is based on hill climbing and provides a near optimal solution within 13% of the minimum cost in most cases by visiting around 10^{-4} of the search space. We ran experiments to test our heuristic and compared it with `FillSlotsFirst` which allocates users to the lower cost per users VMs. Our heuristic algorithm outperformed `FillSlotsFirst` dramatically as shown in the experiments.

References

[1] A. Aldhalaan and D. A. Menascé. Autonomic allocation of communicating virtual machines in hierarchical cloud data centers. In *Cloud and Autonomic Computing (CAC), 2014 Intl. Conf.*, pages 161–171, Sept 2014.

[2] A. Almeida, F. Dantas, E. Cavalcante, and T. Batista. A branch-and-bound algorithm for autonomic adaptation of multi-cloud applications. In *Cluster, Cloud and Grid Computing, 14th Intl. Symp.*, pages 315–323. ACM/IEEE, 2014.

[3] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. The price is right: towards location-independent costs in datacenters. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 23. ACM, 2011.

[4] E. Casalicchio, D. A. Menascé, and A. Aldhalaan. Autonomic resource provisioning in cloud systems with availability goals. In *Proc. 2013 ACM Cloud and Autonomic Computing Conf.*, CAC '13, pages 1:1–1:10, New York, NY, USA, 2013. ACM.

[5] T. Genez, L. Bittencourt, and E. Madeira. Workflow scheduling for SaaS/PaaS cloud providers considering two SLA levels. In *Network Operations and Management Symp., 2012 IEEE*, pages 906–912, April 2012.

[6] B. Jennings and R. Stadler. Resource management in clouds: Survey and research challenges. *J. Network and Systems Management*, pages 1–53, 2014.

[7] K. Johnson, Y. Wang, R. Calinescu, I. Sommerville, G. Baxter, and J. Tucker. Services2cloud: A framework for revenue analysis of software-as-a-service provisioning. In *Cloud Computing Technology and Science, IEEE 5th Intl. Conf.*, volume 2, pages 144–151, Dec 2013.

[8] D. Maia, R. Santos, and R. Lopes. Business-driven long-term capacity planning for saas applications. *Cloud Computing, IEEE Tr.*, 3(1):10, 2015.

[9] C. Mastroianni, M. Meo, and G. Papuzzo. Probabilistic consolidation of virtual machines in self-organizing cloud data centers. *Cloud Computing, IEEE Tr.*, 1(2):215–228, 2013.

[10] P. Mell and T. Grance. The NIST definition of cloud computing. *NIST special publication*, 800(145):7, 2011.

[11] D. Menascé, V. Almeida, and L. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall, 2004.

[12] R. Mietzner and F. Leymann. Towards provisioning the cloud: On the usage of multi-granularity flows and services to realize a unified provisioning infrastructure for SaaS applications. In *Services-Part I, 2008. IEEE Congress on*, pages 3–10. IEEE, 2008.

[13] C. Papagianni, A. Leivadeas, S. Papavassiliou, V. Maglaris, C. Cervello-Pastor, and A. Monje. On the optimal allocation of virtual resources in cloud computing networks. *Computers, IEEE Tr.*, 62(6):1060–1071, 2013.

[14] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[15] F. Shaikh and D. Patil. Multi-tenant e-commerce based on saas model to minimize it cost. In *Advances in Engineering and Technology Research (ICAETR), 2014 Intl. Conf.*, pages 1–4, Aug 2014.

[16] Y. Tan and C. Xia. An adaptive learning approach for efficient resource provisioning in cloud services. *ACM Sigmetrics Performance Evaluation Review*, 42(4):3–11, 2015.

[17] D. Westermann and C. Momm. Using software performance curves for dependable and cost-efficient service hosting. In *Proc. 2nd Intl. Wkshp. Quality of Service-Oriented Software Systems*, QUASOSS '10, pages 3:1–3:6, New York, NY, USA, 2010. ACM.

[18] L. Wu, S. K. Garg, and R. Buyya. SLA-based resource allocation for software as a service provider (saas) in cloud computing environments. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 195–204. IEEE, 2011.

[19] Z. Xiao, W. Song, and Q. Chen. Dynamic resource allocation using virtual machines for cloud computing environment. *Parallel and Distributed Systems, IEEE Tr.*, 24(6):1107–1117, 2013.

[20] Y. Ying, R. Birke, C. Wang, L. Chen, and G. Natarajan. On energy-aware allocation and execution for batch and interactive MapReduce. *ACM Sigmetrics Performance Evaluation Review*, 42(4):22–30, 2015.

[21] Z. Yusoh and M. Tang. Composite SaaS placement and resource optimization in cloud computing using evolutionary algorithms. In *Cloud Computing, 2012 IEEE 5th Intl. Conf.*, pages 590–597, June 2012.

[22] Z. Yusoh and M. Tang. A penalty-based grouping genetic algorithm for multiple composite saas components clustering in cloud. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE Intl. Conf.*, pages 1396–1401, Oct 2012.

[23] Z. Zhang, L. Cherkasova, and B. Loo. Exploiting cloud heterogeneity to optimize performance and cost of MapReduce processing. *ACM Sigmetrics Performance Evaluation Review*, 42(4):38–50, 2015.