Design and Evaluation of Automatic Workflow Scaling Algorithms for Multi-Tenant SaaS

Ankita Atrey[1], Hendrik Moens[1], Gregory Van Seghbroeck[1], Bruno Volckaert[1], and Filip De Turck[1]
[1] *INTEC-IBCN-iMinds, Ghent University, Gaston Crommenlaan, 9050, Gent, Belgium*
{*ankita.atrey, hendrik.moens, gregory.vanseghbroeck, bruno.volckaert, Filip.DeTurck*}*@intec.ugent.be*

Keywords: Cloud multi-tenancy, Cloud simulation, Cloud resource provisioning, SLA provisioning.

Abstract: Current Cloud software development efforts to come up with novel Software-as-a-Service (SaaS) applications are, just like traditional software development, usually no longer built from scratch. Instead more and more Cloud developers are opting to use multiple existing components and integrate them in their application workflow. Scaling the resulting application up or down, depending on user/tenant load, in order to keep the SLA, no longer becomes an issue of scaling resources for a single service, rather results in a complex problem of scaling all individual service endpoints in the workflow, depending on their monitored runtime behavior. In this paper, we propose and evaluate algorithms through CloudSim for automatic and runtime scaling of such multi-tenant SaaS workflows. Our results on time-varying workloads show that the proposed algorithms are *effective* and produce the best *cost-quality* trade-off while keeping *Service Level Agreements (SLAs)* in line. Empirically, the *proactive* algorithm with careful parameter tuning always *meets the SLAs* while only suffering a *marginal* increase in average *cost* per service component of $\approx 5-8\%$ over our baseline *passive* algorithm, which, although provides the least cost, suffers from prolonged violation of service component SLAs.

1 INTRODUCTION

Cloud computing has redefined the way in which computing is perceived today and its use has increased over the years. Clouds offer many benefits, but achieving *scalability* and *quality* of such services while achieving all *Service Level Agreements (SLAs)* at a minimal cost is challenging.

With the increased use of multi-tenancy (W.Tsai and Zhong, 2014), (sharing of virtualised resources among multiple users, thereby increasing concurrency and lowering virtualisation overhead) and rising popularity of cloud services for large user bases (e.g. Dropbox, Office365 etc.), correctly and automatically scaling these services to deal with current user demand becomes very important. Besides this, the majority of the SaaS developers no longer develop their applications from scratch, but utilize specialized existing (cloud-based) services in an application workflow. Examples of such re-usable service endpoints are payment services, authorisation services, cloud monitoring / profiling services, feedback services, etc.

In this work we assume that the resulting SaaS product/application workflow will have to cater to a Service Level Agreement (SLA) with regards to e.g. response time. The issue we face when trying to solve the SLA requirements of a multi-tenant workflow is that each service component in such a workflow, can have different SLA requirements, and can behave differently when put under multi-tenant load. Manual management of the upscaling or downscaling (i.e. assigning more or fewer resources on offering that service) of specific workflow components based on monitored behavior may solve this, but is a process which would have to be done continuously and, due to the manual intervention, would be error-prone. Therefore, in this work, we present an automated multi-tenant workflow SLA monitoring framework, and algorithms which can automatically propose scaling specific service components up or down based on their current SLA compliance.

The use case we are investigating in particular (Fig. 1), is an elastic multi-tenant online collaborative meeting room tool, consisting of workflows which can, for ease of understanding, be simplified to three service components namely encoders, decoders and transcoders. Here, an interactive professional meeting service is offered to a group of employees situated across the globe. Every stream consists of an encoder, potentially a transcoder and a decoder, all of which have different multi-tenant SLA requirements in order to provide a flawless service (no A/V inter-

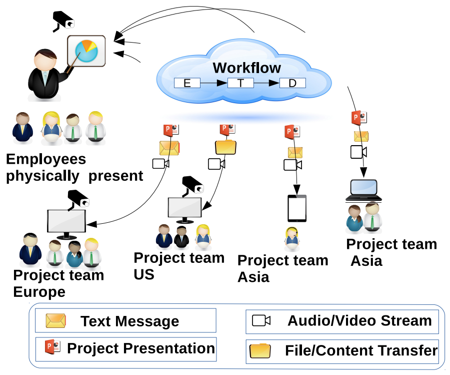

Figure 1: Use-case: An online collaborative meeting room with participants from across the globe.

ruptions, no stuttering, no connection loss, etc.). Note that users can join or leave these meetings at any point in time, leading to large fluctuations in terms of number of tenants currently using the system.

The algorithms proposed in this paper deal with the situation where resource scaling must be performed for each individual service component of the overall application workflow. The runtime behavior of each component is captured by a generic monitoring mechanism, and is used to keep track of how this component is behaving based on its current tenant load and assigned resources. Thus, we intend to automatically intervene if the load on a particular service is becoming too high for it to keep its SLA. Likewise, resource assignment should also be automatically downscaled to save on resources and/or budget.

The rest of this paper is structured as follows. Related work is presented in Section 2, while Section 3 present the problem statement. Following this, Section 4 introduces the SLA monitoring-based resource provisioning algorithms. Section 5 discusses the evaluation setup after which Section 6 discusses the CloudSim evaluation results of the proposed heuristic algorithms. Finally, Section 7 concludes.

2 RELATED WORK

A lot of work has been performed with regards to Cloud resource provisioning strategies for IaaS (Infrastructure as service), PaaS (Platform as a service) and SaaS (Software as a service) providers. Moreover, research on multi-tenancy in cloud applications (Guo et al., 2007) with SLA-driven simulations (Antonescu and Braun, 2014) is not uncommon today.

(Espadasa et al., 2013) have focused on under and over-provisioning of resources in SaaS and its influence on cost-effectiveness. In their work, a multi-tenant based resource allocation model has been designed. Research done by (Bellenger et al., 2011) discussed semi-automatic and automatic scaling. The authors provide an overview of the pros and cons of semi-automatic (users are forced to balance requesting more resources to avoid under-provisioning versus releasing resources to avoid over-provisioning) versus automatic scaling (users follow workloads).

User satisfaction is the key concern of cloud services, which, in certain situations, can be adversely affected by SLA violations. (Morshedlou and Meybodi, 2014) state that SLA violations depend on some user characteristics, and eventually define two types of user characteristics to reduce the impact of SLA violation. Another interesting work (L. Wu and Buyya, 2011), deals with algorithms for automated resource provisioning of SaaS services based on their SLA. This work was further extended to develop a method for admission control (L. Wu and Buyya, 2012) of user requests, thus facilitating prevention of additional user requests from being accepted which in turn would lead to violating the SLA of the service. In continuation to this, (L. Wu et al., 2014) also focused on Customer Satisfaction Level (CSL) which depends on the SLA violations. To improve CSL and reduce SLA violations, various algorithms are designed based on resource reservation and request rescheduling. The work presented in this paper differs from all the works discussed above, owing to our focus on *workflows* of service endpoints, each with independent runtime behavior, but contributing to the overall application workflow's SLA adherence as well.

Various other studies (Taheri et al., 2014) (Glitho, 2011) show that auto-scaling for multimedia services is an actively studied topic of research. A recent work by (Soltanian et al., 2015) lay their focus on a very specific sub-problem of scaling media services. This work differs from the work presented in this paper as our focus is to make *generic* and *robust* algorithms for the entire service workflow spectrum.

3 PROBLEM DESCRIPTION

This section presents a concise model of multi-tenant, multi-component SaaS workflows, with an introduction of its basic concepts followed by a formal description of the *ARP-M* (Automatic Resource Provisioning under Multi-tenancy) problem. Table 1 summarizes the notations used in the rest of the paper.

The basis of the issue at hand is the observation that cloud-based *SaaS* applications currently are, a lot of times, built as *workflows* of multiple existing services (albeit with the necessary custom glue code to tie all of them together). In multi-tenant usage scenarios, as is mostly the case when dealing with SaaS

Table 1: **Summary of the notations used.**

Item	Definition
$\mathcal{V}$	The pool of VMs; $\forall i, V_i \in \mathcal{V}$.
$\mathcal{W}$	Set of workflow requests; $\forall j, W_j \in \mathcal{W}$.
$\mathcal{D}(t)$	Time varying distribution for varying worfklow requests.
C_{kj}	A service component for the workflow W_j; $\forall k, C_{kj} \in W_j$.
$SLA_{status}^{C_{kj}}$	The status (binary) of the SLA of C_{kj}, i.e. met or broken.
$\mathcal{N}_{running}^{i}$	Number of components currently running on VM V_i.
$\mathcal{N}_{max}^{i}$	Maximum number of components allowed on VM V_i.
τ	Parameter controlling how quickly the *Proactive* algorithm intervenes in terms of scaling resources up or down.
$T_{reserve}^{V_i}$	Time required for reservation of a new VM V_i.
$T_{migrate}$	Time required for migrating a service component to V_i.
$P_{reserve}^{V_i}$	Penalty incurred due to reservation of a new VM V_i.
$P_{migrate}$	Penalty incurred owing to migrating components to V_i.

applications, the performance and scalability of each of these workflow service components can behave differently when compared to the others, yet all of them have an impact on the overall performance and scalability of the SaaS application/workflow.

We hence define an application as a workflow $W_j \in \mathcal{W}$ consisting of one or more components $C_{kj} \in W_j$ (see Fig. 2), each of which having a separate SLA agreement which defines a.o. minimal resource requirements (processing power, memory, storage, etc.) in order to work according to its specifications. Service components in such an application workflow, pass their data along the workflow edges to the next service component. In the case of streaming workflows, components receive streaming data from the workflow components which serve as input, while they themselves stream their output data to the workflow components following their execution. Again as an example, if the workflow in Fig. 2 would be representing a streaming workflow, C_{21} would continuously send/stream output data to C_{31} and C_{41}, who in turn process that input data and stream it to C_{51}. All service endpoints are hence processing in parallel.

In this paper, and given our use case of online collaborative audio/video meetings, we will focus on streaming workflows, which process data as long as the meeting is ongoing. It is important to note that this type of workflow, for this type of use cases, does not benefit from assigning more resources to them than required, as one cannot *'speed up'* the meeting. As long as the performance SLAs of the constituent services are met, the meeting service will perform as envisioned for those participating in it (i.e. overallocating resources will not lead to shorter meeting durations). To model this, the tenant requests follow a time-varying distribution $\mathcal{D}(t)$.

We call the problem of automatically provisioning resources for multi-tenant SaaS applications as *Automatic Resource Provisioning under Multi-tenancy (ARP-M)* problem; defined formally as follows.

Problem. *Given a VM pool $\mathcal{V}$, a set of streaming workflow requests $\mathcal{W}$ following a distribution $\mathcal{D}(t)$,*

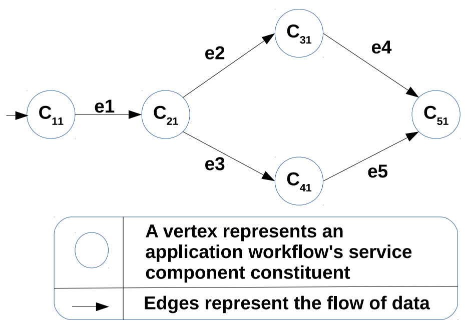
C11
e1
C21
e2
C31
e4
C51
e3
C41
e5
A vertex represents an application workflow's service component constituent
Edges represent the flow of data

Figure 2: An application workflow W_1 composed of multiple service components and inter-component data flows.

and the maximum number of requests allowed on each VM $\mathcal{N}_{max}^{i} \mid \forall V_i \in \mathcal{V}$, perform automatic resource provisioning to keep the SLAs, $SLA_{status} = false$, for all the workflow components $C_{kj} \mid \forall k, C_{kj} \in W_j, \forall j, W_j \in \mathcal{W}$, while retaining high cost-efficiency and quality of service for multi-tenant SaaS.

4 RESOURCE PROVISIONING ALGORITHMS

In this section, we describe our proposed monitoring driven resource provisioning heuristics for meeting the *SLAs* while maintaining high *cost efficiency*. As mentioned in Sec. 3, the SLAs of the workflows depend on the SLAs/run-time behavior of each of its constituent service components. This is characterized by the number of component instances $\mathcal{N}_{running}^{i}$, simultaneously running on the respective provisioned resources $\forall V_i \in \mathcal{V}$ and the maximum number of component instances $\mathcal{N}_{max}^{i}$ that can be served by the resources reserved for the components at any given time. Additionally, the number of the tenant requests follows a time-varying distribution $\mathcal{D}(t)$.

Next, we briefly describe the building blocks of our algorithms: (a) a SLA Monitoring module and (b) a VM Allocation module.

- **SLA Monitoring:** Each component C_{kj} of a workflow $W_j \in \mathcal{W}$ running on a VM $V_i \in \mathcal{V}$, is associated with a binary variable, SLA_{status}, which assumes the value of *False* if SLAs are met and *True* otherwise. Mathematically,

$$SLA_{status} = \begin{cases} false, & \text{if } \mathcal{N}_{running}^{i} \leq \mathcal{N}_{max}^{i} \\ true, & \text{otherwise} \end{cases}$$

  Keeping track of the SLA status for each component (whether it is broken or not) is the main task of this module. This monitoring capability plays a central role in the design of the more involved proposed heuristics, i.e., the *reactive* algorithm (Alg. 3) and the *proactive* algorithm (Alg. 4).

- **VM Allocation:** This module facilitates on demand creation of new VM instances based on a specific VM template from the pool of VMs $\mathcal{V}$. The VM allocations under the *passive* algorithm are performed in the beginning, and remain fixed throughout, whereas are continuously updated for the *reactive* and the *proactive* algorithms.

4.1 Workflow Deployment Algorithm

Algorithm 1 describes the pseudo-code for the deployment of streaming workflows. As mentioned in Sec. 3, the user/tenant requests follow a time-varying distribution $\mathcal{D}(t)$. Now, if the incoming requests at time $t+1$ are greater than those at time t (line 7), then additional workflows are created and assigned to the VMs (depending on the resource provisioning algorithms, lines 8–13), otherwise the additional workflows are canceled and the corresponding resources on the VMs that were hosting these workflows freed (lines 14–24). Note that, irrespective of the *resource provisioning* algorithm, each of the service components $C_{kj} \mid 1 \leq k \leq |W_j|$, of the incoming user/tenant workflow requests W_j, are assigned to separate VMs $V_i \mid 1 \leq i \leq |W_j|$, that are currently accepting requests, from the VM pool $\mathcal{V}$. Once these VMs reach their capacity, then depending upon the protocols of the resource provisioning algorithms new VMs are either reserved / not reserved and the pointer to the currently active VMs altered/unchanged respectively.

4.2 Passive Algorithm

In this algorithm, all the resources with different properties (storage, CPU, memory etc.) are reserved in the beginning of the application session. Note that in this algorithm, no new VM reservations happen even if $\forall V_i \in$ pre-reserved $\mathcal{V}$ the capacity is reached, i.e. $\mathcal{N}^i_{running} > \mathcal{N}^i_{max}$, in which case the SLAs violate.

This algorithm will achieve good results in terms of *Cost* and keeping *SLAs*, only if the request rate is near-constant and the number of requests can fit in the pre-reserved resources. The moment more requests arrive, the SLAs will start to violate and remain violated. The cost, however, will naturally remain fixed.

4.3 Reactive Algorithm

Contrary to the *passive* algorithm, here new VM reservations are triggered once the number of components $\mathcal{N}^i_{running}$ running on a VM V_i exceeds its maximum permissible limit $\mathcal{N}^i_{max}$ (line 6). If the workflow request W_j triggered the reservation process, then l ($1 \leq l \leq |W_j|$) new VMs are instantiated from the VM

Algorithm 1 Workflow Deployment Algorithm

```
Require: $\mathcal{V}$, $\mathcal{N}^i_{max} \mid \forall V_i \in \mathcal{V}$, $\mathcal{W} \sim \mathcal{D}(t), provisionType, \tau$
Ensure: $SLA_{status}, AvgCost, Cost$
 1: $numRunning \leftarrow 0$
 2: for each $V_i \in \mathcal{V}$ do
 3:     $\mathcal{N}^i_{running} \leftarrow 0$
 4: for $t = 0$ to $t_{max}$ do
 5:     $SLA_{status} \leftarrow false, AvgCost \leftarrow 0, Cost \leftarrow 0$
 6:     $\mathcal{W}_t \sim \mathcal{D}(t)$; $numDeploy \leftarrow |\mathcal{W}_t| - numRunning$
 7:     if $numDeploy \geq 0$ then
 8:         if $provisionType = Passive$ then
 9:             $PassiveDeploy(\mathcal{W}_t, \mathcal{V})$
10:         else if $provisionType = Reactive$ then
11:             $ReactiveDeploy(\mathcal{W}_t, \mathcal{V})$
12:         else
13:             $ProactiveDeploy(\mathcal{W}_t, \mathcal{V}, \tau)$
14:     else
15:         for each $W_j \in \mathcal{W}_t$ do
16:             $Cost_j \leftarrow 0$
17:             for each $C_{kj} \in W_j$ do
18:                 Cancel $C_{kj}$ and free its resources on $V_i$
19:                 $\mathcal{N}^i_{running} \leftarrow \mathcal{N}^i_{running} - 1$
20:                 $Cost_j \leftarrow Cost_j - (M_i + C_i + S_i)$
21:                 if $\mathcal{N}^i_{running} \leq \mathcal{N}^i_{max}$ then
22:                     $SLA_{status} \leftarrow false$
23:             $Cost \leftarrow Cost + Cost_j$
24:             $AvgCost \leftarrow AvgCost + Cost_j/|W_j|$
25:         $AvgCost \leftarrow AvgCost/|\mathcal{W}_t|$
26:     $numRunning \leftarrow numRunning + numDeploy$
```

Algorithm 2 Passive Algorithm

```
Require: $\mathcal{V}$, $\mathcal{N}^i_{max} \mid \forall V_i \in \mathcal{V}$, $\mathcal{W} \sim \mathcal{D}(t)$
Ensure: $SLA_{status}, AvgCost, Cost$
 1: procedure PASSIVEDEPLOY($\mathcal{W}_t, \mathcal{V}$)
 2:     for each $W_j \in \mathcal{W}_t$ do
 3:         $Cost_j \leftarrow 0$
 4:         for each $C_{kj} \in W_j$ do
 5:             Deploy $C_{kj}$ on a pre-reserved VM $V_i$
 6:             $\mathcal{N}^i_{running} \leftarrow \mathcal{N}^i_{running} + 1$
 7:             if $\mathcal{N}^i_{running} \leq \mathcal{N}^i_{max}$ then
 8:                 $Cost_j \leftarrow Cost_j + M_i + C_i + S_i$
 9:             else
10:                 $SLA_{status} \leftarrow true$
11:         $Cost \leftarrow Cost + Cost_j$
12:         $AvgCost \leftarrow AvgCost + Cost_j/|W_j|$
13:     $AvgCost \leftarrow AvgCost/|\mathcal{W}_t|$
```

and-above the usual VM utilization costs (line 13).

This algorithm reacts to the detection of violation in SLAs, and thus would suffer from small episodes of SLA violations. Although the SLAs would be met for a large portion of the time, there will be a surge in costs (owing to penalties) when the SLAs are broken.

4.4 Proactive Algorithm

In this algorithm, the *SLA monitoring module* continuously monitors the number of service components

Algorithm 3 Reactive Algorithm

```
Require: $\mathcal{V}$, $\mathcal{N}^i_{max} \mid \forall V_i \in \mathcal{V}$, $\mathcal{W} \sim \mathcal{D}(t)$
Ensure: $SLA_{status}, Cost, AvgCost, AvgPen, AvgSLABrkD$
 1: $SLABrkD \leftarrow 0, AvgSLABrkD \leftarrow 0$
 2: procedure ReactiveDeploy($\mathcal{W}_t, \mathcal{V}$)
 3:     for each $W_j \in \mathcal{W}_t$ do
 4:         $Cost_j \leftarrow 0, Pen_j \leftarrow 0$
 5:         for each $C_{kj} \in W_j$ do
 6:             if $\mathcal{N}^i_{running} + 1 > \mathcal{N}^i_{max}$ then
 7:                 $SLA_{status} \leftarrow true$
 8:                 Identify VM $V_l$, with $\mathcal{N}^l_{running} < \mathcal{N}^l_{max}$
 9:                 Deploy $C_{kj}$ on VM $V_l$
10:                 $\mathcal{N}^l_{running} \leftarrow \mathcal{N}^l_{running} + 1$
11:                 $Cost_j \leftarrow Cost_j + M_l + C_l + S_l$
12:                 $SLABrkD \leftarrow SLABrkD + T^{V_l}_{reserve} + T_{migrate}$
13:                 $Pen_j \leftarrow Pen_j + P^{V_l}_{reserve} + P_{migrate}$
14:             else
15:                 $\mathcal{N}^i_{running} \leftarrow \mathcal{N}^i_{running} + 1$
16:                 $Cost_j \leftarrow Cost_j + M_i + C_i + S_i$
17:         $Cost \leftarrow Cost + Cost_j$; $Penalty \leftarrow Penalty + Pen_j$
18:         $AvgCost \leftarrow AvgCost + Cost_j/|W_j|$
19:         $AvgPen \leftarrow AvgPen + Pen_j/|W_j|$
20:         $AvgSLABrkD \leftarrow SLABrkD/|W_j|$
21:     $AvgCost \leftarrow AvgCost/|\mathcal{W}_t|$; $AvgPen \leftarrow AvgPen/|\mathcal{W}_t|$
22:     $AvgSLABrkD \leftarrow AvgSLABrkD/|\mathcal{W}_t|$
```

Algorithm 4 Proactive Algorithm

```
Require: $\mathcal{V}$, $\mathcal{N}^i_{max} \mid \forall V_i \in \mathcal{V}$, $\mathcal{W} \sim \mathcal{D}(t)$
Ensure: $SLA_{status}, Cost, AvgCost, AvgPen, AvgSLABrkD$
 1: $SLABrkD \leftarrow 0, AvgSLABrkD \leftarrow 0$
 2: procedure ProactiveDeploy($\mathcal{W}_t, \mathcal{V}, \tau$)
 3:     for each $W_j \in \mathcal{W}_t$ do
 4:         $Cost_j \leftarrow 0, Pen_j \leftarrow 0$
 5:         for each $C_{kj} \in W_j$ do
 6:             $\mathcal{N}^i_{running} \leftarrow \mathcal{N}^i_{running} + 1$
 7:             if $\mathcal{N}^i_{running} = \lfloor \tau . \mathcal{N}^i_{max} \rfloor + 1$ then
 8:                 Identify VM $V_l$, with $\mathcal{N}^l_{running} < \mathcal{N}^l_{max}$
 9:                 $StartVM^{V_l} \leftarrow t$
10:             if $\mathcal{N}^i_{running} > \mathcal{N}^i_{max}$ then
11:                 if $t - StartVM^{V_l} < T^{V_l}_{reserve} + T_{migrate}$ then
12:                     $SLA_{status} \leftarrow true$
13:                     $extraDelay \leftarrow T^{V_l}_{reserve} + T_{migrate} - t + StartVM^{V_l}$
14:                     $SLABrkD \leftarrow SLABrkD + extraDelay$
15:                     $Pen_j \leftarrow Pen_j + \frac{extraDelay}{T^{V_l}_{reserve} + T_{migrate}}(P_{reserve} + P_{migrate})$
16:                 Deploy $C_{kj}$ on VM $V_l$
17:                 $\mathcal{N}^l_{running} \leftarrow \mathcal{N}^l_{running} + 1$; $\mathcal{N}^i_{running} \leftarrow \mathcal{N}^i_{running} - 1$
18:                 $Cost_j \leftarrow Cost_j + M_l + C_l + S_l$
19:             else
20:                 $Cost_j \leftarrow Cost_j + M_i + C_i + S_i$
21:         $AvgCost \leftarrow AvgCost + Cost_j/|W_j|$
22:         $AvgPen \leftarrow AvgPen + Pen_j/|W_j|$
23:         $AvgSLABrkD \leftarrow SLABrkD/|W_j|$
24:     $AvgCost \leftarrow AvgCost/|\mathcal{W}_t|$, $AvgPen \leftarrow AvgPen/|\mathcal{W}_t|$,
        $AvgSLABrkD \leftarrow AvgSLABrkD/|\mathcal{W}_t|$
```

$\mathcal{N}^i_{running}$ and checks how far this is from the maximum permissible limit $\mathcal{N}^i_{max}$, for each VM $V_i \in \mathcal{V}$. To address the limitations mentioned in the *reactive* algorithm the *proactive* algorithm incorporates the use of a parameter τ. The parameter τ facilitates the reservation of a new VM V_l and the migration of the service components from V_i to V_l, to be performed while there is still room for more components to be executed on the VM V_i without breaking the SLAs.

Using this algorithm, the SLAs of all the components remain broken for the time required to reserve new VMs and migrate them from the old VM to the new one respectively, *discounting* the time duration corresponding to the start of the reservation process and the time instant at which the SLA actually got violated. Thus, with a careful selection of τ, $T_{reserve} + T_{migrate}$ would get subsumed by the difference in the time instant at which the SLAs actually got violated and the time instant at which the reservation process was triggered. This will result in the SLAs to be always met while the waiting time on VMs that need to be started will also be 0. If the parameter τ is too low, additional VMs will be reserved rapidly which will in turn drive up the cost. Likewise, if the parameter τ is too high, we will spend some extra time to instantiate new VMs. Similar to Alg. 3 penalties are added over-and-above the usual VM utilization costs. Note that since we preach maximum resource utilization, although new VM reservations are triggered once the above condition is met, the service components are migrated only after the VMs currently running them are fully utilized.

5 EVALUATION SETUP

5.1 Media Workflows

The media workflow illustrated in Fig. 3, represents a *streaming* workflow with three components namely encoder, transcoder and decoder. In *streaming* workflows service components continuously receive streaming data from other components which serve as their input, while they themselves stream their output data to other workflow components following their execution. Note that even though much more elaborate workflows exist, this particular workflow has been chosen to showcase the strength of the presented algorithms in an easy-to-grasp manner.

5.2 Evaluation Scenario

The media workflow discussed in the previous section is instantiated by multiple users/tenants and submitted to the *CloudSim* simulator. To showcase our simulation results, we assign the streaming workflow requests ($\mathcal{D}(t)$) to follow a normal distribution. We

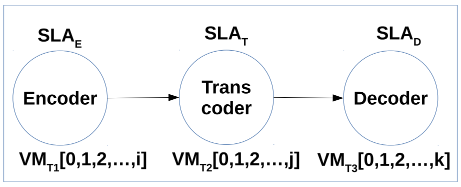

Figure 3: Media workflow with three components: Encoder, Transcoder and Decoder. Each component has a personal SLA and runs on a VM, made available from a VM pool.

generate 200 user requests following a normal distribution, with the time 12 noon set as the mean and 3.5 hours to be the standard deviation. In the beginning of the day requests come in slowly but gradually these requests increase and reach a peak during the mid day.

For each user/tenant request a new instance of the workflow W_j is created and the constituent service components $C_{kj}, \forall k \mid C_{kj} \in W_j$ are provisioned on different VMs V_i, available from the VM pool $\mathcal{V}$ (the choice of VM and how this VM pool grows / shrinks is driven depending on the choice of the algorithm).

5.3 Evaluation Metrics

We consider the following metrics:

- **SLA Status:** The SLA status for each service component C_{kj}, of each instance of a workflow W_j, running on a VM V_i, is defined as a binary variable which assumes the value of $false$ if the SLAs are met, and $true$ otherwise.
- **Average SLA Break Duration:** The average SLA break duration is defined as the amount of time the SLA_{status} of a service component is broken over its runtime duration on average. Thus, for a simulation with w workflow requests $W_j \in \mathcal{W} \mid 1 \leq j \leq w$, c service components each $C_{kj} \in W_j \mid 1 \leq k \leq c$, and $T^{C_{kj}}_{slabreak}$ being the duration for which the SLAs remain broken for a component C_{kj}, we mathematically state the following:
$$\frac{1}{w}\Big(\sum_{j=1}^{w}\big(\frac{1}{c}\sum_{k=1}^{c}(T^{C_{kj}}_{slabreak})\big)\Big) \quad (1)$$
- **VM Cost:** The VM cost is defined as the sum of all costs related to resource usage when running the components of streaming workflows. Thus, for a simulation with w workflow requests, each one with c components, and M_k, S_k, C_k, representing, memory, storage and CPU costs respectively for a component C_k, we mathematically state the VM cost and the average VM cost as follows:
$$\sum_{j=1}^{w}\big(\sum_{k=1}^{c}(M_k + S_k + C_k)\big) \quad (2)$$
$$\frac{1}{w}\Big(\sum_{j=1}^{w}\big(\frac{1}{c}\sum_{k=1}^{c}(M_k + S_k + C_k)\big)\Big) \quad (3)$$

Table 2: **Parameterized VM Templates.**

Template	Storage	CPU	RAM	Monthly Cost
Template$_{01}$	4 GB	40 MIPS	128 GB	\$3.94
Template$_{02}$	8 GB	80 MIPS	256 GB	\$7.88

- **Penalty:** The extra cost incurred over-and-above the normal resource utilization costs accounts for the incurred penalty. The penalty is mainly due to the side-effects of breaking SLAs, and includes the cost spent on components (in SLA break state), while waiting for (1) a new VM reservation $P_{reserve}$ and (2) migration of components from one VM to another $P_{migrate}$. We mathematically state the Penalty and the average Penalty as follows:
$$\sum_{j=1}^{w}\big(\sum_{k=1}^{c}(P_{reserve\,k} + P_{migrate_k})\big) \quad (4)$$
$$\frac{1}{w}\Big(\sum_{j=1}^{w}\big(\frac{1}{c}\sum_{k=1}^{c}(P_{reserve\,k} + P_{migrate_k})\big)\Big) \quad (5)$$

5.4 CloudSim Extensions

The proposed algorithms are implemented and evaluated using the *CloudSim* (Calheiros et al., 2011) event based simulator. To showcase the effectiveness of our algorithms under the proposed evaluation scenario, we implemented the following extensions:

- **SLA Monitor**: For each instantiation of the media workflow, the monitoring module checks the SLAs of all the service components (encoder, transcoder and decoder), to see whether they hold under the current deployment scenarios. The SLAs of various components are continuously monitored by a *Monitor* event, to facilitate triggering of certain actions based on a threshold τ.
- **Resource Provisioner**: The resource provisioning module has been extended to implement all monitoring based multi-tenant resource provisioning algorithms (as defined in Sec. 4).
- **Request Generator**: extends the *Cloudlet* class to support real-time *streaming* workflows and generates user/tenant workflow requests based on a normal distribution until a client event stops it.

6 EVALUATION RESULTS

As mentioned in Sec. 4, the time required to reserve new VMs differs significantly from the time required to migrate one component from an existing VM instance to another. To this end, we define two variables, $T^{V_i}_{reserve}$ and $T_{migrate}$, that determine the duration for instantiating/reserving new VMs and the duration for migrating components to existing VMs respectively. For the simulations, the values of $T^{V_i}_{reserve}$

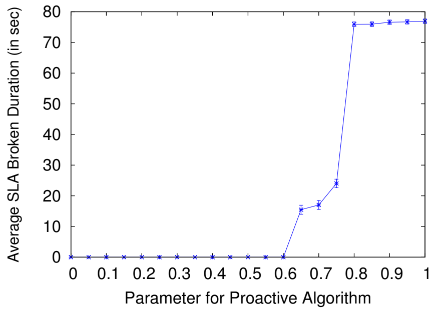

Figure 4: Variation in the average SLA break duration versus τ for the Proactive algorithm.

and $T_{migrate}$ were defined as uniform distributions between [60s,75s] and [0.5s,2s] respectively.

The costs for the VM templates used, were parameterized based on the Amazon EC2 image *c*3.8*x*large, with a monthly price of 1.680 to provide 32 Million instructions per second (MIPS), 60 GB of RAM and 320 GB of storage. This cost was divided equally between secondary-storage, main-memory and CPU, and the converted unit prices (per MB/hour and MI/hour) were used to calculate the costs for the VM templates used in this paper. The computed costs for each VM template are mentioned in Table 2.

All the simulations were executed using the CloudSim simulator and the proposed extensions, on an Intel(R) Core i5 4-core machine with 1.7 GHz CPU and 8 GB RAM running Linux Ubuntu 15.04.

We first analyze the results obtained under the *proactive* algorithm with the variation in the parameter τ from $0 \rightarrow 1$. It is evident from Fig. 4 that the SLAs of the components are met for $\tau \leq 0.6$. Once the value of τ crosses 0.6, the average SLA break duration starts increasing. This increase is at first gradual till $\tau \leq 0.75$, after which it starts increasing rapidly.

The average penalty incurred, due to the time required for a new VM reservation $T^{V_i}_{reserve}$ ($P^{V_i}_{reserve}$) and for migration of components $T_{migrate}$ ($P_{migrate}$), during the time when SLAs for the components are broken, portrays a similar pattern as depicted by the SLA break duration (Fig. 4). Thus, with respect to minimizing the SLA violation duration and minimizing the average penalty, the range $0 \leq \tau \leq 0.75$ is considered to be optimal.

Fig. 5 presents the results on average cost incurred with varying τ. The red line represents the average VM cost (Eq. 3), which is almost constant with the variation in τ. The green line represents the average penalty incurred due to SLA violation of one or more endpoints, which has already been analyzed above. The penalty incurred due to the proactive reservations of VMs is depicted by the blue line. It is evident from Fig. 5 that this penalty linearly decreases with increasing τ, with its maximum value when $\tau = 0$ and minimum value when $\tau = 1$. The total cost represented by the purple line, is the sum of the VM cost and the two penalties discussed above. It is evident that the total cost first linearly decreases till $\tau = 0.65$, becomes almost constant till $\tau = 0.75$ and then starts to increase rapidly with the increase in τ. Thus, with respect to minimizing the total cost, $0.65 \leq \tau \leq 0.75$ serves as the optimal range for parameter τ.

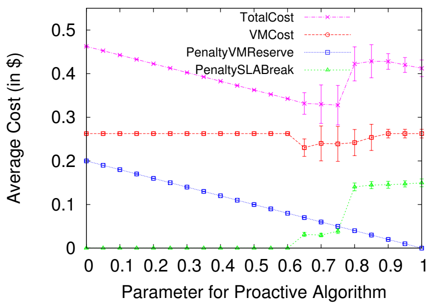

Figure 5: Variation in the average cost versus τ for the Proactive algorithm.

Next, we compare the total and the average cost for the proposed algorithms – (1) *passive*, (2) *reactive* and (3) *proactive*. Note that the *proactive* algorithm will use $\tau = 0.60$ for all the following comparisons.

Fig. 6 portrays the variation in total cost with the time of day for the three proposed algorithms. It is not surprising to see that the *passive* algorithm possesses the least total cost. Since, no new VM reservations happen, even when $\mathcal{N}^i_{running}$ exceeds $\mathcal{N}^i_{max}$, the VM costs are kept at a bare minimum. On the other hand, since new VM reservations happen for both *reactive* and *proactive* algorithms, the costs are naturally higher here. The costs for the *reactive* algorithm are higher, at certain times, when compared to the *proactive* algorithm owing to the penalties occurred due to breaking the SLAs. Note that, although the costs incurred by the *reactive* and the *proactive* algorithm are higher when compared to that of the *passive* algorithm, this cost is warranted (i.e. it is not due to sub-optimal utilization of VM resources, but should be seen as a necessity in attaining the workflow SLAs for the given number of multi-tenant requests).

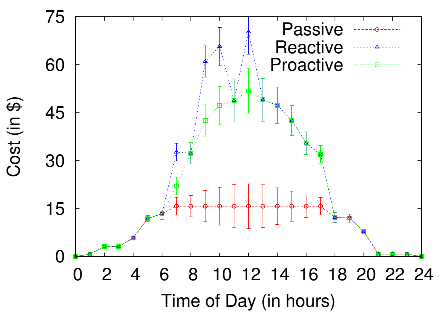

Figure 6: Comparing variation in the total cost versus the time of day for the passive, reactive and proactive algorithm.

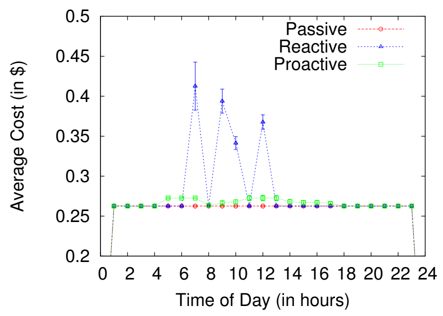

Figure 7: Comparing variation in the average cost versus the time of day for the passive, reactive and proactive algorithm.

To substantiate the above argument, we also compare the variation in the average cost, with the time of day for the proposed algorithms. It is evident from Fig. 7 that the average cost of all the three algorithms are almost similar at majority of the time instances. Note that the portrayed costs also include the penalties incurred, if any. At certain instances, the average cost of the *reactive* algorithm is the highest, which is the result of the penalties incurred due to the VM reservation process starting only after the SLAs are broken. Since the *proactive* algorithm, triggers the new VM reservation process prior to detecting violation in the SLAs, the penalties incurred for this algorithm are significantly lower when compared to that of the *reactive* algorithm. The only penalty incurred on the *proactive* algorithm is due to the pre-reservation of VMs, which is optimized for $\tau = 0.60$ as discussed above. The costs for the *proactive* algorithm are almost similar to that of the *passive* algorithm, while being marginally higher only at certain times.

To summarize, the *proactive* algorithm with τ in the range $0.60 \leq \tau \leq 0.75$ serves as the best possible trade-off for minimizing the costs while also keeping the SLAs of the components in line.

7 CONCLUSIONS

In this paper we have proposed algorithms that allow automatic scaling of SLA-bound SaaS workflows consisting of multiple (SaaS) service endpoints based on monitored application multi-tenancy (where client request rates can highly fluctuate based on the time of day). The effectiveness of these algorithms was demonstrated using a simulated use case of a professional cloud-based A/V collaboration service. These algorithms kept track of the SLAs of each workflow component and, for the most advanced *proactive* algorithm, reserved new VMs before SLAs were to be broken, thus, providing the best possible trade-off between *cost efficiency* and *quality-of-service*.

Future work will see us extending our algorithms with more advanced issues like dealing with service robustness and resource/connection failures. Another line of research will involve mapping SaaS applications and workflows to the TOSCA standard, thus enabling their standardization and use of management plans (Binz et al., 2014) for automatic scaling.

ACKNOWLEDGEMENTS

This research is partly funded by the IWT SBO DeCoMAdS project.

REFERENCES

Antonescu, A. F. and Braun, T. (2014). Sla-driven simulation of multi-tenant scalable cloud-distributed enterprise information system. In *ARMS-CC*.

Bellenger, D., Bertram, J., Budina, A., Koschel, A., Pfänder, B., and Serowy, C. (2011). Scaling in cloud environments. In *WSEAS*.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2014). *Advanced Web Services*, chapter TOSCA: Portable Automated Deployment and Management of Cloud Applications.

Calheiros, R. N., Ranjan, R., Beloglazov, A., Rose, C. A. F. D., and Buyya, R. (2011). Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1).

Espadasa, J., Molinab, A., Jimneza, G., Molinab, M., Ramreza, R., and Conchaa, D. (2013). A tenant-based resource allocation model for scaling software-as-a-service applications over cloud computing infrastructures. *FGCS*, 29(1).

Glitho, R. H. (2011). Cloud-based multimedia conferencing: Business model, research agenda, state-of-the-art. In *CEC*.

Guo, C. J., Sun, W., Huang, Y., and Gao, B. (2007). A framework for native multi-tenancy application development and management. In *CEC*.

L. Wu, S. K. G. and Buyya, R. (2011). Sla-based resource allocation for software as a service provider (saas) in cloud computing environments. In *CCGrid*.

L. Wu, S. K. G. and Buyya, R. (2012). Sla-based admission control for a software-as-a-service provider in cloud computing environments. *JCSS*, 78(5).

L. Wu, S. K. G., Versteeg, S., and Buyya, R. (2014). Sla-based resource provisioning for hosted software-as-a-service applications in cloud computing environments. *IEEE TSC*, 7(3).

Morshedlou, H. and Meybodi, M. (2014). Decreasing impact of sla violations:a proactive resource allocation approach for cloud computing environments. *IEEE TCC*, 2(2).

Soltanian, A., Salahuddin, M. A., Elbiaze, H., and Glitho, R. (2015). A resource allocation mechanism for video mixing as a cloud computing service in multimedia conferencing applications. In *CNSM*.

Taheri, F., George, J., Belqasmi, F., Kara, N., and Glitho, R. (2014). A cloud infrastructure for scalable and elastic multimedia conferencing applications. In *CNSM*.

W.Tsai and Zhong, P. (2014). Multi-tenancy and sub-tenancy architecture in software-as-a-service (saas). In *SOSE*.