Cost-aware scalability of applications in public clouds

Daniel Moldovan, Hong-Linh Truong, Schahram Dustdar
Distributed Systems Group, TU Wien
E-mail: {d.moldovan, truong, dustdar}@dsg.tuwien.ac.at

***Abstract*—Scalable applications deployed in public clouds can be built from a combination of custom software components and public cloud services. To meet performance and/or cost requirements, such applications can scale-out/in their components during run-time. When higher performance is required, new component instances can be deployed on newly allocated cloud services (e.g., virtual machines). When the instances are no longer needed, their services can be deallocated to decrease cost. However, public cloud services are usually billed over predefined time and/or usage intervals, e.g., per hour, per GB of I/O. Thus, it might not be cost efficient to scale-in public cloud applications at any moment in time, without considering their billing cycles.**

In this work we aid developers of scalable applications for public clouds to monitor their costs, and develop cost-aware scalability controllers. We introduce a model for capturing the pricing schemes of cloud services. Based on the model we determine and evaluate the application's costs depending on its used cloud services and their billing cycles. We further evaluate cost efficiency of cloud applications, analyzing which application component is cost efficient to deallocate and when. We evaluate our approach on a scalable platform for IoT, deployed in Flexiant[1], one of the leading European public cloud providers. We show that cost-aware scalability can achieve higher application stability and performance, while reducing its operation costs.

***Keywords*-Cloud, Scalability, Run-Time Control, Cost Efficiency, Elasticity**

I. Introduction

Run-time costs evaluation is required for understanding and controlling scalable cloud applications running in public clouds [1]. Currently, applications deployed in public clouds can be built from a combination of custom software components, and public cloud services. Such services span from virtual infrastructure, to image storage, monitoring, and platform services such as message queues. Many cloud applications are scalable, capable of automatic/manual scale-out/in according to particular requirements [2]. Depending on requirements, scale-out actions allocate additional cloud services to run new instances of scalable application components. In turn, scale-in actions deallocate the added instances, reducing the number of used cloud services, and the application's running costs.

While applications are scaled-out due to performance requirements, cost is the main driver for scale-in [3], [4]. Cost-aware scalability controllers consider the costs of different types of cloud services used by the application, rather than just manipulate their number [5]. However, cost of scalable applications is complex, some services having multiple cost elements. E.g., a VM service could be billed both every hour, and separately per each GB of generated I/O. Certain costs can be static, such as costs for reserving a cloud service [6], [7], [8]. Other costs can be dynamic, such as discounts for certain service usage levels. Costs of cloud services can also depend on service combinations. For example, the reservation cost of an Amazon EC2[2] VM service depends both on its type, and its storage configuration, a VM optimized for high I/O costing more than a regular one. Additionally, public cloud services are usually billed over predefined time and/or usage intervals. This means it might not be cost efficient to deallocate such services at any moment in time. For cost efficiency, we must analyze the cloud service's usage w.r.t., its billing cycle. E.g., a cloud service billed per GB of I/O, which has generated 1.5 GB of I/O, can generate another 0.5 GB at no additional cost.

Due to this cost complexity, developers of scalable cloud applications require support for monitoring costs and developing cost-aware scalability controllers. The usage of cloud services employed by different scalable component instances can also diverge in time. Thus, different component instances can be billed differently over time, being more/less cost-efficient to deallocate during scale-in operations. In existing work such as [5], [7], [9], the authors highlight the cost complexity of scalable applications. However, they do not give insight in their cost efficiency, and do not capture all their cost aspects.

In our work we introduce a platform for monitoring costs and analyzing cost efficiency of scalable applications running in public clouds. To this end, we introduce a model for capturing the pricing schemes of cloud services, and define algorithms for evaluating application costs. We further define and evaluate the cost efficiency of cloud applications and their scalable components' instances. Based on their cost efficiency, we recommend to cost-aware scalability controllers which component instance to deallocate and when.

The rest of the paper is structured as follows. Section II presents the motivation and approach. Section III details our costs analysis algorithms. Section IV introduces our cost efficiency concept, and its computation formula. Section V presents our integrated cost analysis platform. Section VI evaluates our approach. Section VII discusses related work. Section VIII concludes the paper and outlines future work.

This work was partially supported by the European Commission in terms of the CELAR FP7 project (FP7-ICT-2011-8 #317790)

[1] www.flexiant.com/

[2] http://aws.amazon.com/ec2/

978-1-5090-1961-8/16 $31.00 © 2016 IEEE
DOI 10.1109/IC2E.2016.23

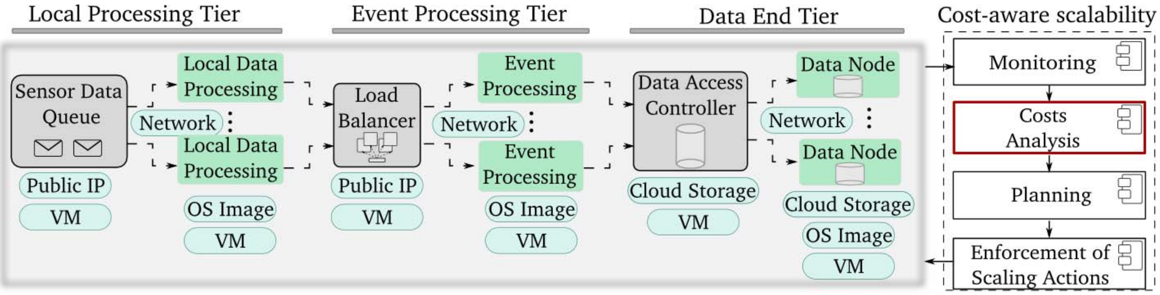

Fig. 1: Motivating Scenario - Cost-aware scalable cloud platform for smart environments

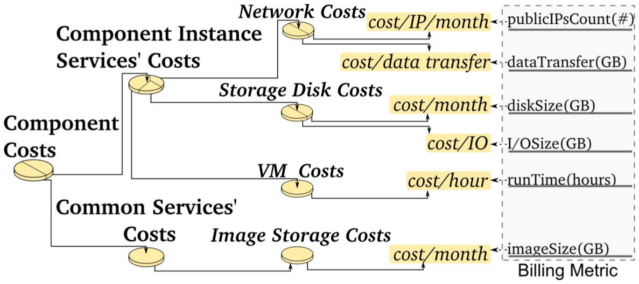

Fig. 2: Example of scalable component costs

II. Motivation and Approach

A. Motivation

It is not cost efficient to scale-in applications running in public clouds at any moment in time. In private clouds, services can be deallocated when they are no longer needed [10], as no billing is involved. However, public cloud services are usually billed over crisp time and/or usage intervals, rounding up the service usage. For example, let us consider two VMs billed per hour and per GB of I/O. If allocated at the same time, and one generates 0.5 GB, and the second 0.9 GB, it is cost efficient to deallocate the second one, as each would be billed for one GB of I/O. If not allocated at the same time, their run-time costs must also be considered. Thus, for cost-aware scalability in public clouds, all costs must be analyzed, ensuring unused but paid for services are not deallocated.

Let us consider a company managing smart environments, relying on sensors sending data for processing to a cloud-based platform (Fig. 1). Sensors are enabled/disabled depending on requirements (e.g., expected data frequency), or special events (e.g., activate water sensors during fire). Thus, the platform is designed as a scalable application, adding/removing at run-time instances of its components. Components are scaled-out by allocating cloud services at run-time, to cope with varying demand, and scaled-in to reduce operating costs. Data is received by a *Sensor Data Queue*, and analyzed by *Local Data Processing* components. It is further sent through a *Load Balancer* to instances of *Event Processing* components, and stored in a distributed data repository.

All application's components use *Virtual Machine (VM)* and *Network* cloud services. The scalable components (depicted in pairs in Fig. 1) use *OS Image* services for storing custom OS images used in allocating new components' instances. The *Data End* tier's components also use high performance *Cloud Storage* services. Finally, the *Sensor Data Queue* and *Load Balancer* use *Public IP* services for exposing their functionality to users. The application's control mechanism enforces performance and costs requirements. It analyzes the application's state, plans and executes scaling actions by adding/removing instances of the application's scalable components, relying on monitoring and costs information. However, costs of scalable applications deployed in public clouds can be very complex. The costs of a scalable component, e.g., the *Data Node* (Fig. 2), can be composed of: (i) the cost for each *Common* service shared by all instances of the component, e.g., OS image; (ii) and costs of all the cloud services used by each component instance, e.g., *VM*, *Storage* or *Network*. Each cost element can be further billed over different cost metrics, which should be considered in cost-aware scalability.

For cost-aware scalability of applications in public clouds, developers and scalability controllers must understand the application's costs, posing several research questions:

- ”How does each application component contribute to the overall application's costs?”, i.e., which component is more expensive, and over which cost elements.
- ”What are the current application costs?”, i.e., the rate at which application components are spending money.
- ”What application component instance is cost efficient to deallocate and when?”, allowing controllers to avoid deallocating services paid in full but underused.

B. Approach

We analyze costs of scalable cloud applications, aiding run-time controllers in deciding what to scale-in and when. Cloud providers are black boxes, providing APIs for allocating/deallocating services on-demand and querying pricing schemes. Scaling actions are enforced by controllers depending on application specifics, interacting with the clouds' APIs.

For achieving cost-aware scalability in public clouds, we develop a platform for monitoring costs and analyzing cost efficiency of scalable applications (Fig. 3), providing.

- A model and mechanism for describing and managing complex pricing schemes of cloud services (①).

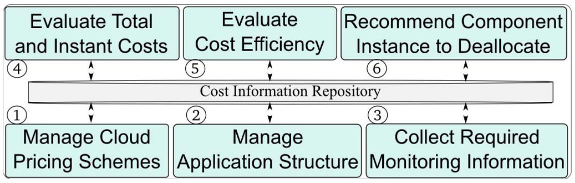
Evaluate Total and Instant Costs
Evaluate Cost Efficiency
Recommend Component Instance to Deallocate
④
⑤
⑥
Cost Information Repository
①
②
③
Manage Cloud Pricing Schemes
Manage Application Structure
Collect Required Monitoring Information

Fig. 3: Approach for cost-aware scalability in public clouds

Billing type	Billing function	Required monitoring information for computing service cost
per Reservation	Fixed	- service allocation time
	per Interval	- service reservation time (allocation/deallocation) - number of allocated service instances
per Usage	Fixed	- current value of the cost billing metric
	per Interval	- current value of the cost billing metric - summed historical values of the billing metric

TABLE I: Billing types of public cloud services

- A mechanism for describing scalable cloud applications and their used cloud services (②).
- A mechanism for collecting cost billing metrics from instances of application components and associating them to the application structure (③).
- A service evaluating the total and instant costs for the application, its components, and used cloud services (④).
- A service evaluating cost efficiency of component instances, providing insight in how much was used from what it has been paid for (⑤).
- A service recommending which application component instance to be deallocated during scale-in operations based on desired cost efficiency, providing support for cost-aware scalability in public clouds (⑥).

III. Evaluating costs of scalable applications

We first capture the pricing schemes of cloud providers. We then determine and evaluate the application's costs, depending on the cloud services used by the application.

A. *Capturing the pricing schemes of public cloud services*

Cloud providers offer multiple types of services, with configuration options under different prices. To capture their pricing schemes, we define a representational model based on previous work [11], centered on the *Cost Element* concept (Fig. 4). Each *Cost Element* has a *type*, either service *Reservation* flat rate (e.g., hourly), or per *Usage* (e.g.,I/O). Cost is computed per billing cycle, over service reservation time (e.g., used hours) or usage *Unit* (e.g., 1 GB I/O), defined over a billing *Metric* (e.g., VM uptime, I/O). Additionally, cost can be specified in intervals, a *costFunction* defining the cost over intervals of measured *billingUnits* over the billing *Metric*. Finally, a *Cost Element* might be applicable only if the service is used in a particular configuration or combination with other services. This is specified with an *applicableIfServiceHas* property, defining which *Resource* and *Quality* properties, or other *Cloud Services* the service should have. Thus, we can describe from fixed costs, to costs if used in conjunction with other services, w.r.t. usage and reservation time.

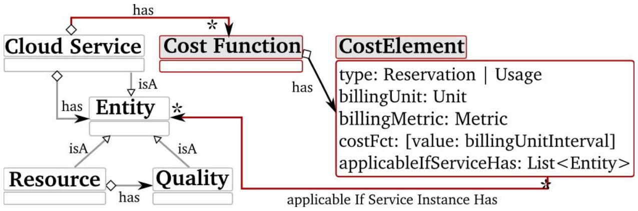
has
*
Cloud Service
Cost Function
CostElement
type: Reservation | Usage
billingUnit: Unit
billingMetric: Metric
costFct: [value: billingUnitInterval]
applicableIfServiceHas: List<Entity>
isA
has
has
Entity
*
isA
isA
Resource
Quality
has
applicable If Service Instance Has
*

Fig. 4: Cost model of public cloud services

During run-time, horizontally scalable components will be instantiated more times, each component instance using cloud services with potentially different configurations. Thus, the application structure and associated cloud services can change at run-time [12], [5], [9]. After any scaling action we determine the applicable *Cost Elements* for the current application structure. For each cloud service used by each component, we search in the service's cloud provider for the complete service description, and applicable cost functions. Then, for each cost element, we verify its applicability conditions, evaluating if the component uses the Resources, Quality, or associated Cloud Services specified in the cost element's description.

When evaluating cost of scalable applications, the information required by different cost billing types must be considered. To classify the billing types of public cloud services (Table I) we analyze main cloud providers: Amazon EC2[3], Flexiant[4], Azure[5], IBM Cloud[6], and Rackspace[7]. Cost can be billed over service reservation time or usage, as a fixed rate (e.g., α\$ per hour), or over an interval (e.g., first x GB of I/O free, next β\$). For computing *Fixed Reservation* cost, we must retrieve from the cloud provider the service instantiation time. For *Interval Reservation* cost we must monitor how many service instances where reserved and for how long, depending if the cost is billed over instance count or time. For *Fixed Usage* cost, we must monitor the billing metric, while for *Interval Usage* cost we need to record the historical usage over the billing metric, to determine the applicable cost interval.

B. *Evaluating cloud application's services' usage and lifetime*

To use the correct pricing interval in cost evaluation, using Algorithm 1 we maintain an updated view over the usage and lifetime of the cloud services employed by the application. Starting from the applicable cost functions, for each cost element, we retrieve its `billing metric` (Line 4). If the element is billed per `Reservation`, we compute how many billing cycles have passed between the last two monitoring intervals (Line 7), and add it in the application usage. If the cost is per `Usage`, we record the total usage over the billing metric (Line 13). Monitored metrics can also be `cumulative`, i.e., they never reset, continuously increasing by adding new monitoring values to the previous ones. With

[3] https://aws.amazon.com/ec2/
[4] https://www.flexiant.com/
[5] https://azure.microsoft.com/
[6] http://www.ibm.com/cloud-computing/
[7] http://www.rackspace.com/

Algorithm 1 Determining usage of cloud services

```
Input: $el : Component|Tier|Application$
Input: $p : previousely\ determined\ usage$
Input: $aC$ : applicable cost scheme
Input: $m : current\ monitoring\ snapshot$
Output: $u : updated\ element\ usage\ snapshot$
 1: function UPDATEELEMENTUSAGE($el, p, m, aC$)
 2:     for s in el.usedServices do
 3:         for ce in aC.GetApplicableCost(el, s) do
 4:             metric = ce.billingMetric
 5:             if ce.type == Reservation then
 6:                 t=GetTimeBetween(m.timestamp,p.timestamp)
 7:                 billingCycles=GetBillingCycles(metric, t)
 8:                 if p.Contains(s) then
 9:                     billingCycles += p.GetLifetime(s,el)
10:                 end if
11:                 u.UpdateLifetime(s,el,billingCycles)
12:             else if ce.type == Usage then
13:                 currVal = m.GetValue(metric,s,el)
14:                 if metric.type == Cumulative then
15:                     u.SetValue(s,el,metric,currVal)
16:                 else
17:                     prevVal = 0
18:                     if p.Contains(s) then
19:                         prevVal = p.GetValue(metric,s,el)
20:                     end if
21:                     usage = prevVal + currVal
22:                     u.SetValue(s,el,metric, usage)
23:                 end if
24:             end if
25:         end for
26:     end for
27:   return u
28: end function
```

such metrics, we use their value as the updated application usage (Line 15). Otherwise, we add their current value to the previous usage (Lines 17-23).

C. Evaluating current and total costs of scalable applications

For cost-aware scalability, developers and controllers require information about the application's current cost, i.e., the billing rate for its current configuration. Such information is crucial in evaluating if the current overall application cost is too high. Thus, we apply Algorithm 2 to evaluate the current costs of cloud applications, based on the latest monitoring information and the application's usage evaluated with Algorithm 1. For each of the cloud services used by the application (Line 4), we analyze each applicable cost element (Line 5). For cost per `Reservation`, we compute the applicable cost based on the element's cost intervals, w.r.t. the lifetime of the used service (Line 9), and store the value directly in the current cost. For cost per `Usage`, we determine the applicable cost value based on the application's usage so far (Line 13). With cumulative metrics we add the cost value to the current cost (Line 15). Otherwise, the latest metric value is multiplied with the cost, and added in the current cost (Lines 17-18).

Algorithm 2 Determining application current costs

```
Input: $el : Component|Tier|Application$
Input: $sU$ : total application usage snapshot
Input: $aC$ : applicable cost scheme
Input: $m : current\ monitoring\ snapshot$
Output: $iC$ : instant costs
 1: function EVALCURRENTCOST($el, sU, aC, m$)
 2:     elementCost=0
 3:     ▷ compute current cost rate for each used service
 4:     for s in el.usedServices do
 5:         for ce in aC.Get(el, s) do
 6:             metric = ce.billingMetric
 7:             if ce.type == Reservation then
 8:                 lifetime = sU.GetLifetime(s,el)
 9:                 cmICost = ce.GetCostForValue(lifetime)
10:                 iC.SetValue(metric, s, el, cmICost)
11:             else if ce.type == Usage then
12:                 metricVal = m.GetValue(metric, s, el)
13:                 compCost=ce.GetCostForValue(metricVal)
14:                 if metric.type == Cumulative then
15:                     iC.SetValue(metric, s, el, compCost)
16:                 else
17:                     cmICost = metricVal * compCost
18:                     iC.SetValue(metric, s, el, cmICost)
19:                 end if
20:             end if
21:             elementCost += cmICost
22:         end for
23:                         ▷ compute hierarchical cost composition
24:         for childEl in el.containedElements do
25:             cChild = EvalCurrentCost(childEl, sU, aC, m)
26:             iC.AddCost(cChild)
27:             elementCost += iC.GetElementCost(childEl)
28:         end for
29:     end for
30:     iC.SetElementCost(elementCost)
31:   return u
32: end function
```

Developers or controllers might be interested only in certain cost information. They could monitor application's cost until a threshold, after which they would be interested in the most expensive components. Thus, we build a hierarchical view over application's cost, computing for each component its current costs by recursively computing the costs of all its used cloud services (Lines 24-28). In turn, the costs of each tier and overall application are evaluated from the costs of its contained components and tiers, and the used cloud services.

We apply a similar algorithm for computing total application costs since its deployment. For this we multiply the total application usage (obtained using Algorithm 1) with the cost element's values, for each of the element's cost intervals.

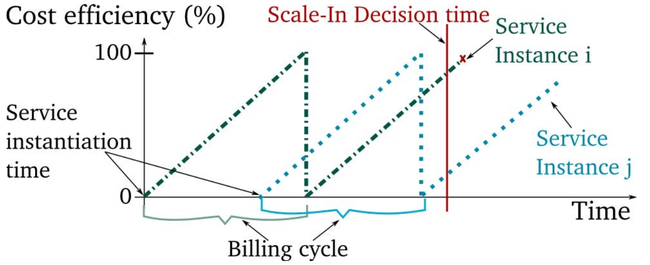

Fig. 5: Service instance cost efficiency w.r.t. billing cycle

IV. Evaluating cost efficiency of scale-in actions

We define a function evaluating the cost efficiency of deallocating application component instances, based on the pricing schemes of the cloud services used by the application.

After multiple allocations/deallocations of cloud services during scaling, the billing cycles of the cloud services used by different application components can become desynchronized. Billing can occur at different points in time, depending on when each component instance was allocated, and on its cloud services' usage (Fig. 5). For increasing the application's cost efficiency, cost-aware control is required, understanding which component is more cost efficient to deallocate, and when.

When scaling-in an application component, a cost-aware controller should analyze which component instance is more cost efficient to deallocate. This depends on the billing cycle of the used cloud services. For maximum cost efficiency, a cost-aware controller should deallocate the component instance having a usage of 100% over all its cost elements (e.g., *Instance i* in Fig. 5). For example, if a component uses cloud services billed per hour and per GB of data, it is cost efficient to deallocate it when it has run for an integer number of hours, and has generated an integer number of GBs. To this end, we define a function E for evaluating the cost efficiency of deallocating an application component instance i, based on the application total usage obtained with Algorithm 1. The function E computes a cost-weighted sum of the instance usage over all its cost elements, both over reservation time and monitored usage, reported to the overall billed cost units:

$$E(i) = \frac{\sum\limits_{s \in i.serv} \sum\limits_{c \in s.cEl} c_{.value} * (s_{.usage}(c_{.metric}) \bmod c_{.cycle}))}{\sum\limits_{s \in i.serv} \sum\limits_{c \in s.cEl} c_{.value}} \tag{1}$$

, where $i.serv$ are the cloud services used by the component instance i; $s.cEl$ are the applicable cost elements for used cloud service s; $c_{.value}$ is the cost value in units of the cost element c (e.g., I/O cost) for one billing cycle $c_{.cycle}$ (e.g., 1 GB, 1 hour); and $s_{.usage}(c_{.metric})$ is the current usage interval over the metric $c_{.metric}$ over which cost is billed.

Weighting the instance usage with the cost value ensures that each cost element has a contribution proportional to its cost to the cost efficiency. Thus, a cost efficiency E of 1 means

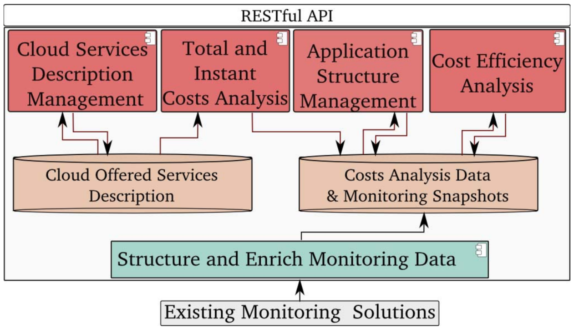

Fig. 6: Platform for analyzing costs of scalable applications

Listing 1: Cloud pricing scheme description fluent API

```
CloudService service = new CloudService().withName("Name")
 .withUuid(UUID).withCategory("Name").withSubcategory("Name")
 .withCostFunction(new CostFunction()
   .withAppliedIfServiceInstanceUses(List<Unit>)
   .withCostElement(new CostElement()
     .withCostMetric(new Metric("name", "unit/time",Type))
     .withBillingInterval(new MetricValue(v), costUnits)
     .withBillingInterval(...
```

that the instance usage is the same as the total billed usage, and by deallocating it, the cost efficiency does not decrease.

V. Costs analysis platform prototype

We implement a costs analysis platform[8] (Fig. 6) for monitoring costs and analyzing cost efficiency of scalable applications, applying our approach from Sections III and IV.

A. Managing pricing schemes of cloud providers

Cloud providers expose their pricing schemes under different mechanisms, from proprietary APIs to plain HTML descriptions. Thus, we provide an easy to use mechanism for describing the cloud services and their cost elements from any cloud provider. To this end we extended MELA [13], a framework for monitoring and analyzing scalable applications. We add a new platform component for managing the description of cloud services, relying on an XML-based representation of the pricing model defined in Section III-A. As manually managing XML descriptions can be difficult, we provide a Java-based Fluent API (Listing 1) for generating them. Cloud services are described using their unique identifiers (UUID), name, category and subcategory, provided by the cloud provider. Their cost is specified as cost functions with applicability restrictions (*applicableIfServiceInstanceUses*). Each cost function can have multiple cost elements with different billing metrics and intervals. Custom adapters for retrieving the pricing scheme of cloud providers can be built using the fluent API to generate the cloud's XML description.

B. Managing the structure of scalable cloud applications

Scalable cloud applications change their structure at runtime by allocating/deallocating cloud services. Thus, we provide a mechanism for updating the application's structure

[8] Prototype and supplement materials are available at http://tuwiendsg.github.io/MELA/costEvaluationService.html

Service			Cost Element				
Category	Subcategory	Service	Billing metric	Billing unit	Billing cycle	Billing interval	Cost units
IaaS	VM	1.0CPU1.0 - 1 CPU 1GB RAM	instance	#	hour	0→Inf.	3
		2.0CPU4.0 - 2 CPU 4GB RAM					10
	Cloud Storage	VM-attached cloud storage	disk size	GB	month	0→Inf.	5
			I/O	GB			2
	Network	Public	data transfer	GB		0→Inf.	5
		Private	instance	#	month		0
	Image	Custom operating system image	snapshot size	GB	month	0→Inf.	5

TABLE II: Subset of described Flexiant pricing scheme

Listing 2: Cloud application structure description fluent API

```
MonitoredElement vm = new MonitoredElement("UUID")
    .withCloudOfferedService(
            new UsedCloudOfferedService()
                .withCloudProviderID("UUID"))
                .withCloudProviderName("Provider")
                .withId("UUID").withInstanceUUID("UUID")
                .withName("ServiceName")
                .withResourceProperties(Map<Property,Value>)
                .withQualityProperties(Map<Property,Value>)
    ).withCloudOfferedService(...
```

and the used cloud services during run-time, relying on the XML representation of the model introduced in [13]. Using this model, a cloud application is represented as a cascading set of `Monitored Elements` representing the application's components, which in turn are grouped in tiers. We extend the model to allow specification of the used cloud services, and provide a fluent API for describing the application's structure (Listing 2). The used cloud service instances are identified by the unique identifier (UUID) of the cloud provider offering the service, the UUID of the service in the provider's service's list, and the UUID of the service's instance. Further, if the same service has different costs for specific configurations, the concrete configuration is specified in terms of resource/quality properties. The description must be updated after each scaling action, to ensure costs analysis consistency. The platform exposes the description to interested stakeholders in XML and JSON formats, and as a tree-based visualization.

C. Collecting and enriching monitoring information

The structure of scalable applications is dynamic, as VMs are created/destroyed dynamically at run-time. If monitoring information is associated only with each VM, it will be lost during scale-in operations. Addressing this, we use MELA [13], which provides a mechanism enriching and associating to the current application structure monitoring information collected from existing monitoring systems. For example, associating a CPU usage metric collected from a VM to the instance of the application component hosted on the VM. Moreover, monitoring data can be enriched by applying aggregation operations over collected metrics (e.g., average CPU usage over all instances of a component), or injecting information using a `set` operation. The metric composition rules can be described directly in XML, or using a fluent API.

As cloud applications use a wide range of monitoring mechanisms, we extend MELA with poll-based data collection adapters for Ganglia[9] and JCatascopia [14]. We also implement a generic push-based adapter, exposing a queue for receiving monitoring information.

D. Evaluating application costs and cost efficiency

The algorithms from Sections III and IV are applied to evaluate the total and instant costs, and cost efficiency of cloud applications. The algorithms use the enriched monitoring information, application structure, and the cloud pricing schemes. Cost analysis information is exposed through RESTful services, and the complete cost decomposition is provided in comma-separated-values (CSV) format. Services are implemented for evaluating the cost efficiency of scaling in one or more instances of an application component, and for recommending which component instance to deallocate w.r.t., to desired cost efficiency. Aiding human users, we provide web-based visualizations for the application's costs, both using tree and pie charts, implemented in D3.js[10].

VI. Cost analysis platform evaluation

We evaluate our costs approach on the scalable cloud platform from Section II (Fig. 1), deployed in Flexiant[11], one of the leading European public cloud providers. The application developer deploys our platform on a separate VM, and uses it to understand the application's costs under expected load. The used application has a shared-nothing architecture, each component using its own cloud services. However, our approach is also applicable on shared architectures, with multiple components sharing a cloud service. In such a case finer-grained monitoring is required to differentiate the service usage generated by different components.

A. Configuring the platform for the target application

First, the application developer describes the pricing scheme of the cloud services offered by Flexiant[12] (Table II), and submits it to our platform. While some services have monthly cost rates, the billing is done per hour, and thus the pricing scheme is specified in hourly rates.

Next, the developer describes the application structure with the used cloud services. As Flexiant provides separate billing for *VM* and *Cloud Storage* services, separate services are specified for each component. Components accessible from outside

[9] http://ganglia.sourceforge.net/

[10] http://d3js.org/

[11] We use Flexiant as an illustrative example in this paper, but similar issues apply to other public cloud providers

[12] www.flexiant.com/2010/04/14/flexiscale-2-0-pricing/

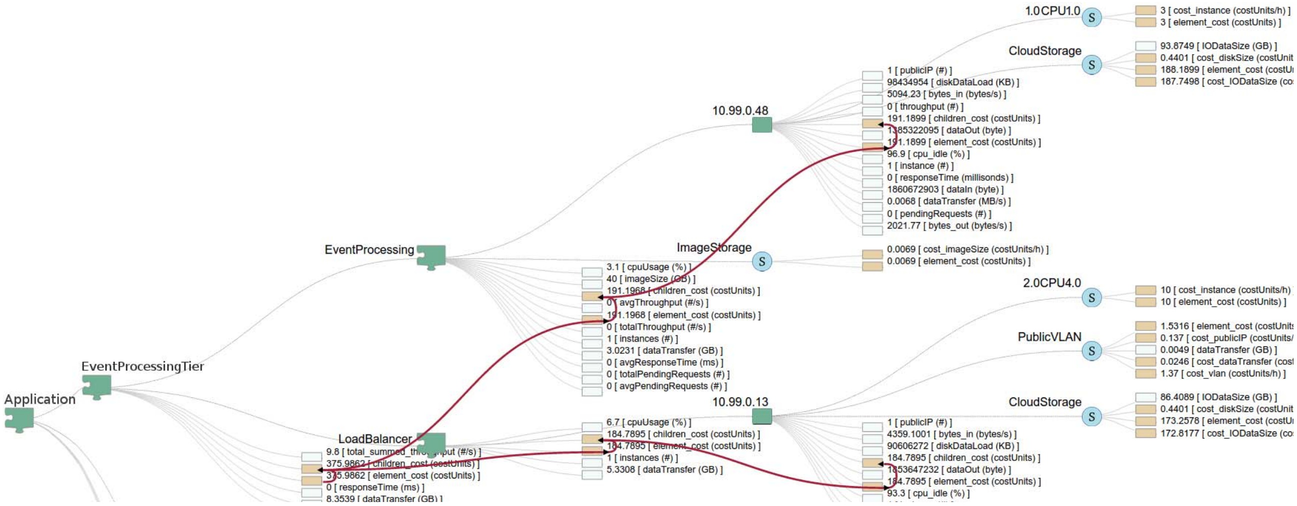
Application
EventProcessingTier
EventProcessing
LoadBalancer
9.8 [total_summed_throughput (#/s)]
375.9862 [children_cost (costUnits)]
375.9862 [element_cost (costUnits)]
0 [responseTime (ms)]
8.3539 [dataTransfer (GB)]
ImageStorage
3.1 [cpuUsage (%)]
40 [imageSize (GB)]
191.1968 [children_cost (costUnits)]
0 [avgThroughput (#/s)]
191.1968 [element_cost (costUnits)]
0 [totalThroughput (#/s)]
1 [instances (#)]
3.0231 [dataTransfer (GB)]
0 [avgResponseTime (ms)]
0 [totalPendingRequests (#)]
0 [avgPendingRequests (#)]
10.99.0.48
1 [publicIP (#)]
98434954 [diskDataLoad (KB)]
5094.23 [bytes_in (bytes/s)]
0 [throughput (#)]
191.1899 [children_cost (costUnits)]
1385322095 [dataOut (byte)]
191.1899 [element_cost (costUnits)]
96.9 [cpu_idle (%)]
1 [instance (#)]
0 [responseTime (millisonds)]
1860672903 [dataIn (byte)]
0.0068 [dataTransfer (MB/s)]
0 [pendingRequests (#)]
2021.77 [bytes_out (bytes/s)]
1.0CPU1.0
3 [cost_instance (costUnits/h)]
3 [element_cost (costUnits)]
CloudStorage
93.8749 [IODataSize (GB)]
0.4401 [cost_diskSize (costUnit
188.1899 [element_cost (costU
187.7498 [cost_IODataSize (co
0.0069 [cost_imageSize (costUnits/h)]
0.0069 [element_cost (costUnits)]
2.0CPU4.0
10 [cost_instance (costUnits/h)
10 [element_cost (costUnits)]
PublicVLAN
1.5316 [element_cost (costUnits
0.137 [cost_publicIP (costUnits/
0.0049 [dataTransfer (GB)]
0.0246 [cost_dataTransfer (cost
1.37 [cost_vlan (costUnits/h)]
10.99.0.13
6.7 [cpuUsage (%)]
184.7895 [children_cost (costUnits)]
184.7895 [element_cost (costUnits)]
1 [instances (#)]
5.3308 [dataTransfer (GB)]
1 [publicIP (#)]
4359.1001 [bytes_in (bytes/s)]
90606272 [diskDataLoad (KB)]
184.7895 [children_cost (costUnits)]
1853647232 [dataOut (byte)]
184.7895 [element_cost (costUnits)]
93.3 [cpu_idle (%)]
CloudStorage
86.4089 [IODataSize (GB)]
0.4401 [cost_diskSize (costUnit
173.2578 [element_cost (costU
172.8177 [cost_IODataSize (co

Fig. 7: Cloud service's costs associated to monitoring information for the Event Processing Tier

the cloud use *Public VLAN* services providing public IPs (e.g., *Load Balancer*). The scalable components use *OS Image Storage* services in instantiating new component instances on top of new *VMs*. The described application's structure is visualized using our platform, Fig. 7 depicting the *Event Processing* tier's structure after the initial cloud deployment (before scaling). Used cloud services are represented with Ⓢ.

Metric resolution for associating the cost billing metrics to the ones collected from the application components is done based on name and measurement unit. Thus, the developer ensures the necessary monitoring information can be collected, i.e. the billing metrics specified by the cost elements captured in Table II are available. The application is configured to expose the required information through our platform's Ganglia[13] adapter, implementing the required plug-ins, such as collecting disk I/O for analyzing IO cost. To evaluate *OS Image Storage* service cost, information hard to monitor directly is required, such as the size of each OS image. We add this information directly in the monitoring data using MELA metric composition rules, creating the image size metrics for each scalable component.

B. Evaluating costs of scalable applications

Using our platform, the developer visualizes the application's monitoring snapshot enriched with cost (Fig. 7). Evaluated cost is represented with darker rectangles. Cost of individual component instances is summed up in overall cost for each component, tier, and the overall service, under a `element_cost` metric (highlighted with thick line). The costs of the used cloud offered services (e.g., *Cloud Storage*) are associated to the component instances running inside virtual machines. The costs of each component instance are further aggregated in the costs of each component (e.g., *Event Processing* component). These in turn are aggregated in the costs of the parent tiers, up to the overall application costs. Thus, cost information can be retrieved at the desired level, from component to application, along monitoring information. This increases the potential applicability of our approach.

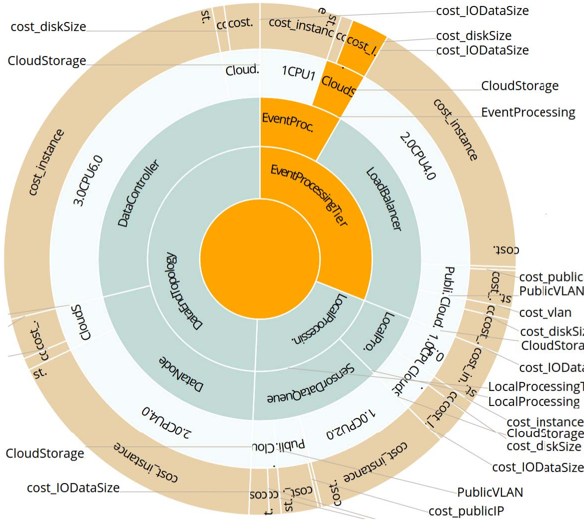

Fig. 8: Application costs composition

What is the cost contribution of each application component and tier? is visualized by our platform in Fig. 8. The application costs are composed of the costs of its tiers, composed from the costs of their components. The cost of individual components consists of the costs of the used instances of *VM* services, and the costs of the *Cloud Storage*, reported both in price per disk size, and per size of read/written data. We also have the costs of additional used services, such as the *Public VLAN*. From the cost proportion of each cost element, the developer understands which components have low cost impact (e.g., *Local Processing* component), and which are

[13] http://ganglia.sourceforge.net/

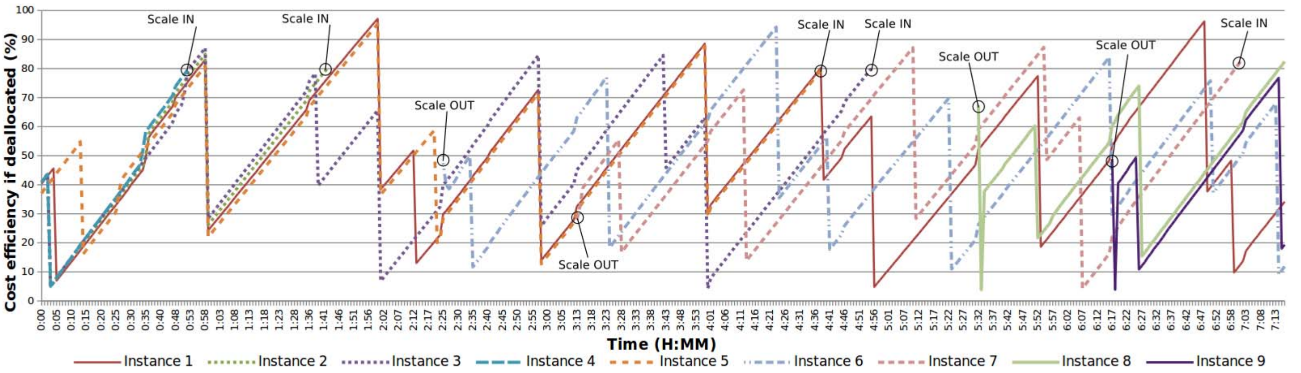

Fig. 9: Cost efficiency variation after multiple scaling actions between instances of the Event Processing component

costly (e.g., *Data Node*). Under the given load, for the *Event Processing*, the *Cloud Storage* cost is roughly 1/3 of the component's cost. The biggest contributor to storage cost is *IO DataSize*, i.e., cost for read/written data. This is not the case for all components, for the *Data Controller*, the *3 CPU 6GB RAM VM* being more expensive than the *Cloud Storage*. Using this information, the developer can reduce the *Event Processing*'s costs by changing its I/O rate, or switching to cloud services or cloud providers with less I/O cost.

C. Comparing cost-aware and cost-agnostic scalability

Further, the developer uses our platform for building a cost-aware scalability controller considering cost efficiency information. While the application has three scalable tiers, in the following we focus on a single component, the *Event Processing* component. This both fully covers the addressed research questions, and increases the evaluation's readability.

We implement a scalability controller supporting different scaling strategies (Table III), for evaluating the application's cost efficiency improvement under cost-aware scalability. We provide two cost-agnostic scaling strategies which do not consider cost efficiency, and two cost-aware relying on our platform. The two cost-agnostic strategies deallocate the *Last* and *First* allocated component instance. The first cost-aware strategy deallocates a component instance and its associated cloud services when its *VM* has run over 90% of its reservation billing cycle, i.e., run over 54 minutes. The second cost-aware strategy deallocates a component instance when its cost efficiency is over 80%. We selected 80% after repeated experiments, noticing that 90% cost efficiency is not always achievable for our application due to the difference in the cost elements' billing cycles. Using such a limit (90%) can actually increase costs, by waiting too much to deallocate services. The *Event Processing* is scaled with each strategy for over 12 hours. To reduce the variables influencing our results, and better evaluate the scaling strategies, we apply a fixed load of 450 sensors per second. Requests are distributed by the *Load Balancer* in a round-robin manner. We start with five component instances, and repeatedly request 2 scale-out followed by 2 scale-in actions, each action being requested every 45 minutes. Requesting the actions in a time interval < 1 hour (the Reservation billing cycle of the used services) we mimic normal behavior of scalable applications which can add and remove component instances anytime. The fixed action periodicity also allows us to compare the efficiency of the scaling strategies under the same conditions.

Strategy type	Deallocating	Strategy description
Cost-aware	w.r.t. Reservation time	- deallocates the component instance if it has run over 90% of reservation cycle
	w.r.t. Cost efficiency	- deallocates the component instance with cost efficiency over 80%
Cost-agnostic	Last allocated	- deallocates the last allocated component instance
	First allocated	- deallocates the first allocated component instance

TABLE III: Evaluated scalability control strategies

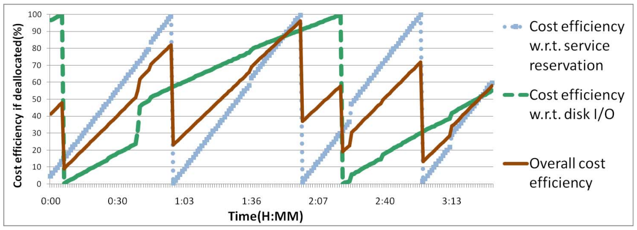

Fig. 10: Decomposed cost efficiency of VM cloud service

What is the application's cost efficiency evolution in time? is answered by analyzing over time the cost efficiency of the *Event Processing* component's instances (Fig. 9). The component is scaled-in with our second cost-aware strategy. Initially, all 5 instances have similar cost efficiency. However, their cost efficiencies start to differ as scaling actions are enforced, due to the different billing cycles of their cost elements. We decompose the billing cycle of the VM service used by the *Event Processing* component in Fig. 10. We show for a single component instance its evaluated cost efficiency obtained by applying the function from Section IV over (i) VM reservation cost, (ii) disk I/O cost, and (iii) all cost elements. From the figure we notice that the disk I/O and the VM reservation cost efficiencies have different cycles, as it roughly

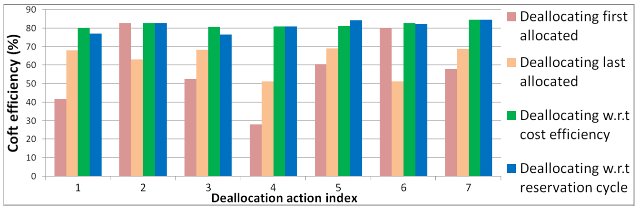

Fig. 11: Scale-in cost efficiency under different strategies

takes 2 reservation periods (1 hour each) to complete a disk billing cycle, i.e., to generate 1GB of I/O. Cloud services could have even more cost elements with separate billing cycles. This highlights the need for cost-aware scalability, understanding which used services are cost efficient to deallocate, and when.

What is the application's cost efficiency improvement under cost-aware scalability? is evaluated by comparing the cost efficiency of each scale-in operation for every control strategy. The efficiency values obtained using our efficiency analysis approach are depicted in Fig. 11. The results show that the cost-aware controller deallocated services with cost efficiency over 80% during enforcement of scalability actions. From Fig. 11 we also notice that except one action, the cost-aware strategy obtained better cost efficiency even compared to the one deallocating components at over 90% of their reservation billing cycle. This highlights that all cost elements need to be considered when scaling cloud applications, as relying only on reservation cycle can lead to cost-inefficient solutions in public clouds with complex pricing schemes. We also noticed that the cost-agnostic strategies obtained lower cost efficiency for their scale-in actions, many times deallocating unused services.

D. Improving application control with cost-aware scalability

Under highly fluctuating load, by opportunistic scaling, controllers can decrease cost efficiency by deallocating services ahead of time, and then allocating new ones to cope with rising demand. While in the above scenario we improve the cost efficiency of scalable applications running in public clouds, the employed uniform control does not capture their dynamicity. To evaluate dynamic scenarios, we select the best cost-agnostic (deallocating last added instance), and the best cost-aware (deallocating w.r.t. overall cost efficiency) strategies from the previous scenario. We implement a scaling generator which issues randomly between 1 and 3 scale-in requests at time intervals between 30 and 60 minutes, mimicking the behavior of real applications which might be scaled-in anytime depending on requirements. We apply the same sequence of scaling requests for each control strategy. Under the cost-agnostic strategy, the controller deallocates component instances as soon as requested. In contrast, the cost-aware controller, when a scale-in is requested, waits until a component instance has reached 80% cost efficiency before deallocating it. Moreover, if a scale-out is to be executed, the cost-aware controller will verify if it is waiting to reach a certain cost efficiency to scale-in. Then, it would cancel the pending scale-in action, instead of executing a new scale-out, as a means of reducing cost.

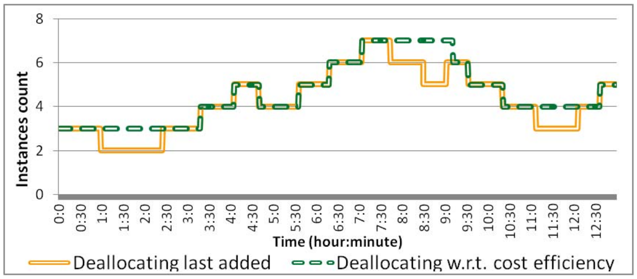

Fig. 12: Number of event processing instances under cost-aware and cost-agnostic scalability

Evaluation criteria	Scalability control		Cost-aware scalability vs. cost-agnostic control
	cost-aware	cost-agnostic	
Scaling actions	10 (6 out, 4 in)	16 (9 out, 7 in)	**37% less actions** = increased application stability
Average no. of instances	4.59	4.24	**8% more instances** = spare resources for load bursts
Total cost (units)	143	153	**7% cost savings** = 3.3 VM hours + 1.5 GB I/O

TABLE IV: Cost-aware scalability control evaluation

What is the benefit of cost-aware scalability in public clouds? is answered in Fig. 12, depicting the number of *Event Processing* component instances under each control strategy. We have run the scenario over 12 hours, and in three cases, the cost-aware strategy avoided unnecessary scale-ins, while the cost-agnostic one deallocated and allocated back component instances. For example, between 7:30 and 9:30, the cost-aware strategy scaled-in just once. In contrast, the cost-agnostic controller performed two scale-in and one scale-out actions. The improvements obtained by the cost-aware controller over the cost-agnostic one are summarized in Table IV. Overall, the cost-aware controller performed with approx. 37% less scaling actions, left 8% more running VMs, all with a cost saving of 7%. Thus, the application is overall more stable, as any scaling can introduce instability due to the cloud infrastructure, and the application adapting to the change in resources. By running more component instances, the application has additional spare resources to accommodate unexpected spikes in load. Additionally, considering the used pricing scheme, the cost savings obtained can be spent to run another *Event Processing* VM for 3 hours, generating approx. 1.5 GB of I/O. Thus, employing cost-aware scalability controllers in public clouds can also increase the performance and stability of cloud applications, while reducing their costs.

VII. Related Work

Andrikopoulos et al. [15] highlight that costs analysis of complex applications is required from their design, to deployment and run-time control. Ramakrishnan et al. [1] underline that cost-benefit evaluation is necessary for e-science clouds, due to the plethora of available cloud services and configuration options. Lorido-Botran et al. [4] survey cloud scaling techniques, Hwang et al. [3] underline that cloud productivity is tied to performance/cost ratio, while Villegas et al. [6] analyze the relationship between provisioning and allocation policies in IaaS clouds and their cost. Truong et al. [16] analyze cost of running scientific applications public clouds, while Sharma et al. [12] focus on cost-based optimization of scalable cloud applications, considering the costs of different types of virtual resources. Douglas et al. [17] estimate the costs of running scientific simulations in public clouds, retrieving cost information from the Amazon API, while Lilienthal [7] computes the optimal resources for hybrid applications running in private-public clouds. Brighen et al. [8] estimate running costs of data intensive applications in clouds, while Guo et al. [18] focus on cost-aware applications deployment in public clouds. While the related work highlights the cost complexity of scalable applications, they do not capture all their costs, and do not give insight in their cost efficiency.

In [5], Mao et al. allocate virtual machines to perform tasks under time deadlines, shutting down VMs when approaching full hour operation. Fernandez et al. [10] scale web applications in heterogeneous cloud infrastructures, executing scale-in actions by releasing resources if the application load exceeds a lower threshold. Pooyan et al. [9] scale-in a cloud service if it has been running a multiple number of hours. Silva et al. [19] introduce a framework for scaling cloud applications for understanding their cost/performance trade off. Hwang et al. [20] focus on cost effective provisioning of cloud services, considering reserved, and on-demand cost, and three cost functions: upfront fee, usage charge, and on-demand cost. Our work can provide to such approaches information about both the costs, and cost efficiency of scalable applications, improving their scalability control.

VIII. Conclusions and Future Work

In our work we focused on aiding developers of scalable applications for public clouds to monitor their costs, and develop cost-aware scalability controllers. We introduced a model for capturing complex pricing schemes of cloud providers, from fixed service costs to costs per multiple services. We defined algorithms for determining the application costs depending on its used cloud services. We defined a function for evaluating cost efficiency of cloud applications, analyzing which application component instance is cost efficient to deallocate and when. We evaluated our approach on a scalable cloud-based data center for smart environments, deployed in Flexiant, one of the leading European public cloud providers.

We have shown that using our approach, developers of scalable cloud applications can determine the applications' dominant cost aspects. We have further shown that cost-aware controllers can achieve higher application stability and performance, while reducing operation costs, compared to cost-agnostic ones. We highlighted that cost-agnostic scaling can lead to deallocating unused services, but paid in full, increasing the application's costs. We have shown that cloud services instantiated at the same time can have different cost efficiencies over time (Fig. 9), highlighting the need to analyze cost when controlling scalable applications in public clouds.

In the future we plan to apply our cost analysis approach on analyzing cost uncertainty in scalable applications, understanding the causes for costs deviations between different instances of the same application component.

References

[1] L. Ramakrishnan, K. R. Jackson, S. Canon, S. Cholia, and J. Shalf, "Defining future platform requirements for e-science clouds," in *Symposium on Cloud Computing (SOCC)*. ACM, 2010, pp. 101–106.

[2] S. Dustdar, Y. Guo, B. Satzger, and H. L. Truong, "Principles of elastic processes," *IEEE Computing*, no. 5, pp. 66–71, 2011.

[3] K. Hwang, X. Bai, Y. Shi, M. Li, W. Chen, and Y. Wu, "Cloud performance modeling and benchmark evaluation of elastic scaling strategies," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2015.

[4] T. Lorido-Botran, J. Miguel-Alonso, and J. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[5] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *International Conference on Grid Computing (GRID)*. IEEE/ACM, Oct 2010, pp. 41–48.

[6] D. Villegas, A. Antoniou, S. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," in *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE/ACM, 2012, pp. 612–619.

[7] M. Lilienthal, "A decision support model for cloud bursting," *Business & Information Systems Engineering*, vol. 5, no. 2, pp. 71–81, 2013.

[8] A. Brighen, L. Bellatreche, H. Slimani, and Z. Faget, "An economical query cost model in the cloud," in *International Conference on Database Systems for Advanced Applications (DASFAA)*. Springer Berlin Heidelberg, 2013, pp. 16–30.

[9] P. Jamshidi, A. Ahmad, and C. Pahl, "Autonomic resource provisioning for cloud-based software," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2014, pp. 95–104.

[10] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *International Conference on Cloud Engineering (IC2E)*. IEEE, March 2014, pp. 195–204.

[11] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, "Quelle - a framework for accelerating the development of elastic systems," in *European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer, 2014.

[12] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 2011, pp. 559–570.

[13] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, "Mela: Monitoring and analyzing elasticity of cloud services," in *International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2013, pp. 80–87.

[14] D. Trihinas, G. Pallis, and M. Dikaiakos, "Jcatascopia: Monitoring elastically adaptive applications in the cloud," in *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE/ACM, May 2014, pp. 226–235.

[15] V. Andrikopoulos, S. Gómez Sáez, F. Leymann, and J. Wettinger, "Optimal distribution of applications in the cloud," in *International Conference on Advanced Information Systems Engineering (CAiSE)*. Springer, 2014.

[16] H. L. Truong and S. Dustdar, "Composable cost estimation and monitoring for computational applications in cloud computing environments," in *International Conference on Computational Science (ICCS)*. Elsevier, 2010, pp. 2175–2184.

[17] G. Douglas, B. Drawert, C. Krintz, and R. Wolski, "Cloudtracker: Using execution provenance to optimize the cost of cloud use," in *Economics of Grids, Clouds, Systems, and Services*. Springer, 2014, pp. 99–113.

[18] T. Guo, U. Sharma, P. Shenoy, T. Wood, and S. Sahu, "Cost-aware cloud bursting for enterprise applications," *ACM Transactions on Internet Technology (TOIT)*, vol. 13, no. 3, pp. 10:1–10:24, May 2014.

[19] M. Silva, M. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. da Silva, "Cloudbench: Experiment automation for cloud environments," in *International Conference on Cloud Engineering (IC2E)*. IEEE, March 2013, pp. 302–311.

[20] R.-H. Hwang, C.-N. Lee, Y.-R. Chen, and D.-J. Zhang-Jian, "Cost optimization of elasticity cloud resource subscription policy," *Transactions on Services Computing*, vol. 7, no. 4, pp. 561–574, Oct 2014.