Multi-level Elasticity Control of Cloud Services*

Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, Schahram Dustdar

Distributed Systems Group, Vienna University of Technology
{e.copil,d.moldovan,truong,dustdar}@dsg.tuwien.ac.at

Abstract. Fine-grained elasticity control of cloud services has to deal with multiple elastic perspectives (quality, cost, and resources). We propose a cloud services elasticity control mechanism that takes into account the service structure for controlling the cloud service elasticity at multiple levels, by firstly defining an abstract composition model for cloud services and enabling, for each level of the composition model, the elasticity control not just in terms of resources, but more importantly, also in terms of quality and cost. Secondly, we define mechanisms for solving conflicting elasticity requirements and generating action plans for elasticity control. Using the defined concepts and mechanisms we develop a runtime system for supporting multiple levels of elasticity control and validate the resulted prototype through real-world experiments.

1 Introduction

Controlling cloud services elasticity in the contemporary view has been targeted by both research and industry. Several authors propose controllers for the automatic scalability/elasticity of entire cloud services [1], just parts of the cloud service (i.e., cloud service data-end) [2], or specific types of cloud services [3]. Cloud providers offer tools for automatic scalability like Amazon's AutoScale[1] or IBM's SmartCloud initiative[2], automatically scaling resources depending on what the user specifies through low-level resource targeted policies. However, these approaches do not control the cloud service on multiple levels taking into consideration the complex service structure, or the multiple dimensions of elasticity (quality, resources, and cost).

Controlling cloud services[3] at multiple abstraction levels may be extremely difficult when cloud services are distributed among several virtual machines. Granularity aware elasticity control enables different situations, e.g., the need for control of the service as a whole when considering black-box cloud services, control of individual service "parts" when the information about these parts

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

[1] http://aws.amazon.com/autoscaling/

[2] http://www.ibm.com/cloud-computing/us/en/index.html

[3] In this paper, *cloud service* refers to the whole cloud application, including all of its own software artifacts, middleware and data, that can be deployed and executed on cloud computing infrastructures.

are available, and control of communication among service parts when possible. Moreover, multiple levels of elasticity control enable easier control of groups of service units which are semantically connected.

In this paper, we propose a system for multi-level cloud services elasticity control by considering the complex structure of the service and supporting a multi-dimensional elasticity composed of cost, quality and resources elasticity. We present the following contributions:

- a generic composition model of cloud services for enabling the fine-grained control aware of the structure of the cloud service
- a fine-grained, multiple levels automatic elasticity control of cloud services
- a runtime engine for a high decision quality and elasticity control aware of the structure of the cloud service which is being controlled

As we demonstrate in our experiments, controlling different parts of the service, while knowing the relation between service parts can be very powerful in the context of cloud computing, where it is advised to have parts of cloud services deployed on different virtual machines for being able to actually profit from on-demand elasticity.

The rest of this paper is organized as follows: Section 2 shows the motivation for this research. Section 3 defines our generic composition model. Section 4 presents our techniques supporting multi-level elasticity control. Section 5 presents experiments. Section 6 discusses our related work while Section 7 concludes the paper and outlines our future work.

2 Motivation

Cloud services are designed in a fashion that they typically use as many as possible resource capabilities from cloud providers. They are deployed and distributed on different virtual machines and consume various types of services offered by cloud providers, possibly from different cloud infrastructures. Therefore, these types of services would have requirements different from the traditional applications, and potentially, they can achieve elasticity not only in terms of resources but also of cost and quality.

In our previous work we have developed SYBL [4], a language for elasticity requirements specification. SYBL enables the user to define various types of elasticity requirements: (i) *monitoring* specifications for specifying which metrics need to be monitored, (ii) *constraints* for specifying acceptable limits for the monitored metrics, and (iii) *strategies* for specifying actions to be taken and conditions in which those actions need to be taken. For showcasing examples and for the experiments section we will be using a cloud service composed of a data service part and a data analysis part, described in detail in Section 5.1. The *cost-related elasticity requirement* specified by, e.g., the service designer, can be that when the total cloud service cost is higher than 800 Euro, a scale-in action should occur for keeping cost in acceptable limits (see Listing 1.1, Strategy $St1$), while a *quality related elasticity requirement* could be that the result from a

data analytics algorithm must reach a certain data accuracy under a certain cost constraint, without caring how many resources are used for executing the code of that algorithm (see Listing 1.1, $Co1$, $Co2$ and $Co3$). The first example is a cloud service level while the second one is a code region level elasticity requirement.

Listing 1.1: SYBL elasticity directives

```
@SYBL_ServiceUnitLevel(strategies=
    "St1: STRATEGY CASE  total_cost>800 Euro : ScaleIn")
@SYBL_CodeRegionLevel(constraints=
   "Co1: CONSTRAINT dataAccuracy>90%;
    Co2: CONSTRAINT dataAccuracy>95% WHEN total_cost>400;
    Co3: CONSTRAINT total_cost<800;"
priorities= "Priority(Co2)>Priority(Co1);
            Priority(Co3)>Priority(Co1);")
```

While elasticity requirements can be specified at different levels, current elasticity control techniques do not support controlling different parts of the cloud service (i.e., elasticity requirement on service unit, on groups of service units, on relationships between service units) and from a multi-dimensional perspective. Controlling the cloud service at multiple levels enable a finer-grained control according to elasticity requirements described by the cloud service designer or developer. On the other hand, multiple levels of elasticity requirements could give rise to conflicts on cross-level or even on the same levels. We propose a cloud service model for the description of both static and runtime information and, based on this model, a solution for overcoming cross-level conflicting elasticity requirements and generating action plans for multi-level elasticity control.

3 Mapping Service Structures To Elasticity Metrics

3.1 Elasticity metrics

Cloud service metrics differ on the service type, the service unit targeted by the metric, or the environment in which the service resides. In general, we could divide these metrics into the following categories:

- *resource-level metrics*, e.g., IO cost, CPU utilization, disk access, latency, memory usage, and number of packets sent or received. Resource level metrics are the most encountered in cloud IaaS APIs.
- *service unit-level metrics*, e.g., number of active threads, used memory, request queue length, response time, performance (which can depend on several resource metrics) and money required to pay. Service units like application server, web server or database server need metrics of a higher level of abstraction than simple resource-level metrics, for easily determining the unit's health or performance.
- *cloud service-level metrics*, e.g., number of users, downtime, average response time, average number of users/day, average cost per user, cloud service availability and cost. Going higher into the abstraction level, when evaluating

the performance of the cloud service one usually considers metrics like cloud service response time, number of users per day in the detriment of low-level metrics like CPU utilization or memory usage.

In elasticity control, these metrics can be associated to different cloud service parts (e.g., the whole cloud service, service unit or a group of service units). Usually, metrics from higher levels (e.g., cloud service level) aggregate metrics from lower levels. For instance, the cost metric can be used when monitoring the cloud services for control at different hierarchical levels. The cost metric as part of the resource metrics category can refer to cost for IOs or virtual machine cost, while cost of a service unit is aggregated over resource cost but can also entail other costs like communication among different virtual machines which host the same service unit, different licensing costs (when using PaaS). The highest level of abstraction, the cloud service level cost for the entire cloud service would aggregate over the underlying costs.

3.2 Abstracting cloud services

For obtaining highly granular control of cloud services and being aware of what service unit is being controlled, a model for structuring service-related information is needed. In multi-level elasticity control such model bridges the gap between the information about the cloud service/system as seen by the cloud user or developer, which mainly regards high level information like costs and overall quality, and the data needed for accurately controlling the cloud service which is generally lower level information.

Our proposed model shown in Figure 1 is generic, aiming at supporting the structure of different kinds of cloud services (e.g., queue-based applications, web applications or batch jobs). The abstract model has the form of a graph, with various types of relationships and nodes, representing both static and run-time description of the cloud service:

- *Cloud Service*, e.g., is a game, a web application, or a scientific workflow application. The cloud service represents the entire application or system, and can be further decomposed into service topologies and service units. The term cloud service that we choose to use here is in accordance with existent cloud architectures and standards (e.g., IBM [5] and TOSCA [6]).
- *Service Unit* [7], e.g., is a database node, or a load balancer. The service unit represents any kind of modules or individual services offering computation and data capabilities.
- *Service Topology*, e.g., is a tier of a cloud service (i.e., business tier or data tier), or a part of a workflow. A cloud service topology represents a group of service units that are semantically connected and that have elasticity capabilities as a group.
- *Code Region*, e.g., is a data analytics algorithm, or a computation intensive sequence of code. Within a service unit, a code region represents a particular sequence of code for which the user can have elasticity requirements.

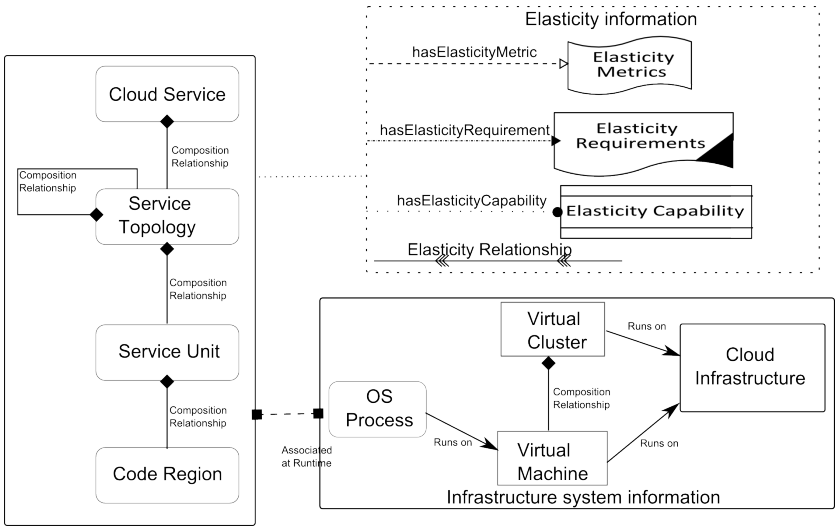

Fig. 1: Cloud service abstraction model

- *OS Process*, e.g., is a web server process. OS processes can be associated either with code regions or with service units.
- *Elasticity Metric*, e.g., is cost vs. performance, cost vs. throughput, or cost vs. availability. Metrics targeted by elasticity requirements or lower-level metrics that are used for computing targeted metrics. Elasticity metrics can be associated with any cloud service part (e.g., service unit, service topology, or code region).
- *Elasticity Requirement*, e.g., is "the cost should not increase by more than 20% when the quality increases by up to 20%". These requirements can be specified through SYBL and can be associated with any cloud service part.
- *Elasticity Capability*, e.g., is the elastic reconfiguration of the data service topology for higher availability, or the elastic creation of new processing jobs for a map-reduce application.
- *Elasticity Relationship*, e.g., is any connection between any two cloud service parts, which can be annotated with elasticity requirements. We choose using the *relationship* term for being in accordance with cloud service specification standards (e.g., TOSCA [6]).

We do not assume that when describing elasticity requirements (e.g., by the service developer) all the abstraction levels of the model have to be targeted or that all the relationships need to be described. For instance, in a compute-intensive service unit when the result from one specific algorithm is extremely important and the developer needs to make sure that the algorithm will not fail due to low performance s/he may mention what are the appropriate quality parameters under which the algorithm functions as expected. In our work, the cloud service model can either be specified directly by the service stakehold-

ers (e.g., service developer, designer, manager or product owner) according to our abstraction model or it can be obtained obtained by analyzing application structures, e.g., described in TOSCA [6].

3.3 Runtime dependency graph of elastic cloud services

In order to describe the cloud service during runtime, a runtime elastic dependency graph is used. This dynamic graph captures all the information about the abstract model and runtime information like elasticity metrics, requirements and deployment topology during runtime and is constructed by the control system. Figure 2 shows how the runtime dependency graph is constructed. If we take the example of a Web service (the left side of Figure 2),the cloud user views his/her Web service as a set of services (in this case `Service C1`, `Service C2`, and `Service C3`), some of them grouped together for monitoring purposes (in this case `Service Group` which consists of `Service C1` and `Service C3`). The metrics targeted in users elasticity requirements in this stage are high level metrics, referring to the quality, cost and resources of services, of groups of services or even of the entire Web service.

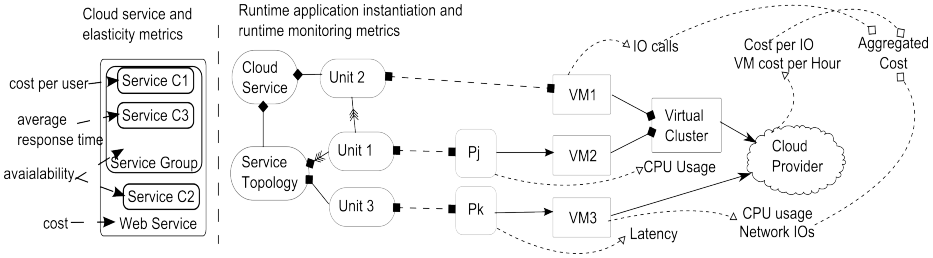

Fig. 2: Constructing runtime dependency graph

At runtime, the dependency graph is constructed (right part of the figure). Service instances are deployed on virtual machines, in different virtual clusters or even different cloud providers, being viewed by the runtime control system through the light of our model (e.g., `Service C1` becomes `Service Unit1`, possibly having more than one instances deployed on more than one virtual machines), and the accessible metrics are low level ones. For the system to control such service it needs to know how to aggregate metrics for obtaining the higher level ones which are targeted by the user, and how services are linked together for having the capacity to properly control them. For bridging this information gap between the user and runtime perspective we use the dependency graph, which facilitates the control system of the cloud service to take control actions considering the complete description of the cloud service.

The runtime dependency graph is associated with metrics information from two views. Firstly, it is associated with metrics from the service user perspective

that can view the cloud service as a Web service and has a high level view concerning metrics. Next, the graph is associated with metrics from the control system perspective which views cloud services in a uniform manner using our abstract model and has an initial low level view concerning metrics, composed of information provided by cloud APIs and monitoring tools. These two views on metrics are mapped by our elasticity control runtime, aggregating low-level metrics for computing higher level ones. For instance, availability at service level would be computed from availability at each service part and the cost is aggregated from static information from the cloud provider on cost per I/O and VM cost, and the run-time service topology and loads.

4 Multi-level Elasticity Control Runtime

4.1 Steps in multi-level elasticity controls

Considering the model of the cloud service described through the abstract model presented in the previous section, we enable multiple levels elasticity control of the described cloud service, based on the flow shown in Figure 3. The elasticity requirements are evaluated and conflicts which may appear among them are resolved. After that, an action plan is generated, consisting of actions which would enable the fulfillment of specified elasticity requirements.

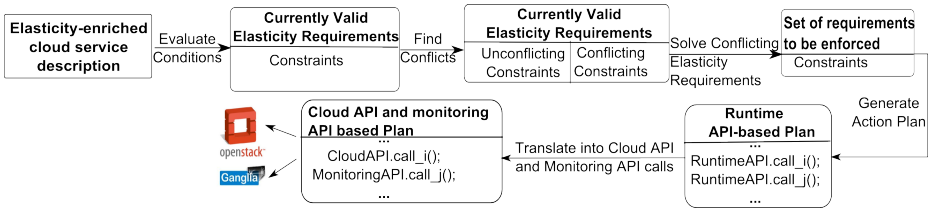

Fig. 3: Elasticity control: from directives to enforced plans

Let us consider a simple example of controlling the entire cloud service, e.g., by the system designer. Let `directive 1` be the `cost` for hosting the entire cloud service should be `less than 40 Euro`, `directive 2` be the `time` that the computation service topology runs is `less than 20 Hours` and `directive 3` be `availability of the cloud service is higher than 99%`. The three directives are not directly conflicting and actions are searched for fulfilling these requirements. Possible actions are, for instance, for the case the running time is higher than 10 hours and the cost is still in acceptable limits to scale-out for the computation service topology, increasing the processing speed. An example of an action plan could be `ActionPlan1=[increaseReplication(1,cassandraTopology),scaleOut(2,computingComp)]`. This action plan would address performance issues for the second

elasticity requirement, and availability issues for the third elasticity requirement. Translated into enforcement API calls, the `increaseReplication` action would consist of calls for adding and configuring a new database node and configuring the cluster for higher replication, while the `scaleOut` action would be the addition of two new machines for computing service and configuration actions necessary for the service unit instances to receive jobs.

4.2 Resolving elasticity requirements conflicts

We identify two types of conflicts: (i) conflicts between elasticity requirements targeting the same abstraction level, and (ii) conflicts which appear between elasticity requirements targeting different abstraction levels. The process of resolving the first type of conflicts is presented in Algorithm 1. Sets of conflicting constraints are identified and a new constraint overriding previous set is added to the dependency graph for each level (lines 3-10).

Algorithm 1 Conflict resolution on the same abstraction level

Input: $graph_i$ - Runtime Dependency Graph
Output: $graph_o$ - Single-level Non-conflicting Runtime Dependency Graph

1: $constraints$ = findAllEnabledConstraints($graph_i$)
2: **for each** l in $cloudServiceAbstractionLevels$ **do**
3: List<List<Constraint>,Level> $confConstraints$= getConflictingConstraints($constraints$,l)
4: $graph_i$.removeConstraints($confConstraints$)
5: List<Constraint> $newGeneratedConstraintsLevel$;
6: **for each** $constraintSet$ in $confConstraints$ **do**
7: $newConstraint$=solveConflictingConstraints($confConstraints$)
8: $newGeneratedConstraintsLevel$.add($newConstraint$)
9: **end for**
10: $graph_i$.addConstraints($newGeneratedConstraintsLevel$)
11: **end forreturn** $graph_o = graph_i$

In the second type of conflict (see Algorithm 2) the constraints from a lower level (i.e., service unit level) are translated into the higher constraint's level (i.e., service topology level), by aggregating metrics considering the dependency graph. Since the problem is reduced to same-level conflicting directives, we use the approach for the same-level conflicting directives and compute a new directive from overlapping conditions. In both (i) and (ii) it can be the case of conflict for directives that are targeting different metrics which influence each other (i.e., cost and availability- when availability increases, the cost increases as well). However, knowing how one metrics' evolution affects the other is a research problem itself which we envision as future work.

For a better illustration we can consider the elasticity requirements in Figure 4, described by the service user/owner, focusing on the execution of service unit

Algorithm 2 Conflict resolution on the different abstraction levels

Input: $graph_i$ - Runtime Dependency Graph
Output: $graph_o$ - Cross-level non-conflicting Runtime Dependency Graph

```
for each level1 in cloudServiceAbstractionLevel do
    for each level2 in cloudServiceAbstractionLevel do
        if level1 ≠ level2 then
            conflictingConstraints.add(getConflictingConstraints(level1,level2))
        end if
    end for
    graph_i.removeConstraints(conflictingConstraints)
    translatedConstraints=translateToHigherLevel(conflictingConstraints)
    graph_i.addConstraints(translatedConstraints)
end forreturn graph_o=solveSameLevelConflictingConstraints(graph_i)
```

`Hadoop Slave` given that `Hadoop Master` is finished doing the "map" part. In this case, `cons1` and `cons 2` are conflicting directives from the same level of abstraction, and we can compute a new directive by choosing as maximum throughput between $x * nbUsers$ and 50000. On the other hand, we can have the second type of conflicts for the $TotalCost$ metric: if we consider that `Hadoop Master` has finished its execution with a cost of 4 Euro, fulfilling `cons7` on the service topology would be in contradiction with `cons5` since the cost is an aggregation of the cost for the service unit `Hadoop Master` and service topology `Data Analysis Topology`. For resolving the conflicts we would need to produce a constraint, `TotalCost<6`, replacing the conflicting ones.

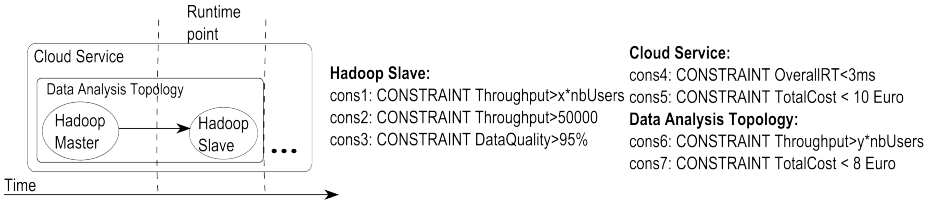

Fig. 4: Cloud service and possible conflicting elasticity requirements

4.3 Generating elasticity control plans

For generating the action plan, we formulate the planning problem as a maximum coverage problem: we need the minimum set of actions which help fulfilling the maximum set of constraints. Since maximum coverage problem is an NP-hard problem, and our research does not target finding the optimal solution for it, we choose the greedy approach which offers an $1 - \frac{1}{e}$ approximation. The greedy approach shown in Algorithm 3 takes as input the dependency graph and returns the sequence of actions necessary for enforcing the constraints.

Algorithm 3 Constraint enforcement

Input: *graph* - Cloud Service Dependency Graph
Output: *ActionPlan*

1: *violatedConstraints* = findAllViolatedConstraints(*model*)
2: **while** size(*violatedConstraints*) > 0 **do**
3: **for each** *level* in *cloudServiceAbstractionLevel* **do**
4: *violatedConstraintsL*=selectConstraints(*violatedConstraints*, *level*)
5: *actionSet*=evaluateLevelEnabledActions(*graph*, *violatedConstraintsL*)
6: *Action*=findAction(*actionSet*) with max(constraints fulfilled - violated)
7: Add action *Action* to *ActionPlan*
8: *violatedConstraints* = findAllViolatedConstraints(*graph*, estimatedActionEffect(*Action*))
9: **end for**
10: **end whilereturn** *ActionPlan*

The main step of the plan generation loop (lines 2-9) consists of finding each time the action for fulfilling the most constraints. For evaluating this, each action has associated the metrics affected and the way in which it affects them (i.e., scale out with `VM of type x increases the cost with 200 Euro`). The number of fulfilled constraints through action enforcement is defined as the difference between the number of constraints enforced and the number of constraints violated. For instance, increasing replication with a factor of one from the example would affect negatively cost and positively performance and availability, resulting in the number of fulfilled constraints through action enforcement one if cost requirement is violated and two otherwise.

4.4 Enforcing action plans

The generated action plan is processed by our Runtime API for creating suitable Cloud API calls realizing the higher-level actions in the action plan. For controlling the elasticity of cloud services, tools monitoring the elasticity and the different types of metrics targeted by the cloud service user (see Section 3.1) are necessary. Although at the moment existing cloud APIs offer only access to low-level resources, elasticity control of cloud services would also impose the existence of cloud APIs which take into account the different levels of metrics or the cloud service structure. For overcoming this situation, we develop our own solution which aggregates low-level metrics for achieving higher level ones, and use existent resource-level control actions for manipulating higher level quality and cost. For instance, the cost of a service unit would be composed of the different types of cost associated to each resource associated with the service unit, like cost depending on the number of virtual machines, cost for intra-unit communication, or I/O cost. The cloud service cost is computed as the sum of service unit cost, inter-unit communication cost, and possible licensing costs.

5 Experiments

We have implemented elasticity control as a service based on SYBL engine [4] for supporting multi-level, cloud service model aware elasticity control of cloud services[4]. We support elasticity controls for cloud services, service units, service topologies, relationships and code regions where elasticity requirements can be specified in XML or through Java Annotations detected at runtime with AspectJ. We tested our prototype on our local cloud running OpenStack[5], using Ganglia[6] and Hyperic SIGAR[7] for monitoring and JClouds[8] for controlling virtual machine instances.

5.1 Experimental cloud service

Our experimental cloud service is a data-oriented application with two parts: a data serving part and a data analysis part. The cloud service has two main service topologies: one data servicing oriented topology and one data analytics oriented topology. The first topology consists of three service units: an YCSB [8] client, a Cassandra[9] controller and several Cassandra data nodes. The YCSB client uses the Yahoo's YCSB benchmark for generating realistic workloads, similar to Facebook status updates, tweets, or photo-tagging services, for the Cassandra cluster. For the YCSB workload we generate a continuous alternation of combinations of the enumerated types of workloads run in parallel. The controller and Casandra data node service units are grouped together into a Cassandra topology, since one may want to monitor and apply actions at cluster level. The second service topology is a simple Hadoop cluster consisting of a Hadoop master and a Hadoop slave to process large data-sets using Mahout machine learning library[10]. The current data analysis service topology processes the latest Wikipedia dump[11] using the Wikipedia Bayes classification example provided by Mahout. The elasticity requirements are specified by the software designer as shown in Figure 5, and control system is deployed on the cloud infrastructure together with the cloud service.

For the current implementation, the number of actions available is limited to scaling in and out at service unit level and at service topology level. For this paper, we make the assumption of knowing how an action taken at current level affects metrics at the lower levels, and aggregate the metrics values for higher levels (i.e., a scale out at service unit level also affects cost at cloud service level). Modeling action effect on metrics is an important research area on itself and is left to future work.

[4] Prototype: `http://dsg.tuwien.ac.at/prototypes/elasticitycontrol/`
[5] http://openstack.org/
[6] http://ganglia.sourceforge.net/
[7] http://hyperic.com/products/sigar
[8] http://jclouds.org/
[9] http://cassandra.apache.org/
[10] http://mahout.apache.org/
[11] www.download.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2

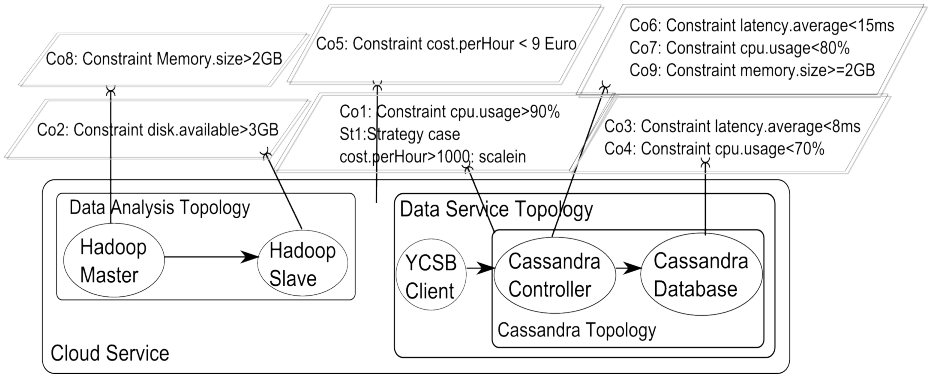

Fig. 5: Current cloud service structure and elasticity directives

5.2 Experimental results

For reflecting the importance of higher level elasticity control in addition to the obvious low level one, Table 1 presents performance and cost data on different Data Service Topology configurations. We assume each virtual machine costs 1 EUR/hour. The first important reason for enabling topology level elasticity is that multiple clusters remove the single-point of failure problem, decreasing the probability of failures which is imminent for the case of highly intensive workloads with a single Cassandra controller.

Configuration	Controllers	DB Nodes	Total execution time	Cost
Config1	1	3	578.4 s	0.48
Config2	1	6	472.1 s	0.91
Config3	2	2	382.4 s	0.42
Config4	3	7	372.2 s	0.72

Table 1: Cost and execution time for `Data Service Topology` units

The behavior of the topology scaled at service unit level differs from the one scaled at service topology level. Although scale out actions at service unit level do manage to increase performance (i.e., `Config2` vs. `Config1` increase in performance of 18.37%), they also enable a considerable cost increase (90 % increase in costs for `Config2` vs `Config1`). In contrast with this action level, a scale out on Cassandra topology is a much better solution, `Config3` offering a performance improvement in time of 33.88% over `Config1`, and a cost improvement of 12.03%. This is due to the fact that more controllers also increase the parallelism of requests, eliminate bottlenecks and facilitate the workload to finish in less time. However, when considering the difference of performance and cost between configurations 3 and 4, it is obvious that the dimension of

the cluster and the number of clusters necessary for meeting the demands of a specific workload are strongly dependent on the workload characteristics and intensity. For improving the user experience we have to take into account both the performance improvement one action may bring, and the manner in which the cost is affected by that action. Therefore, the level at which one action should be enforced strongly depends on the quality metrics for controllers at the higher levels. Figure 6 shows how the elasticity control engine can scale the Data Service Topology both at service unit and at service topology level, when directives shown in Figure 5 require such actions. Considering an initial configuration of a single cluster containing one controller (Cassandra seed) and one simple node, the elasticity control engine takes firstly a scale out action at service unit level for fixing the broken constraint "`Co4`". After that, it takes a new scale out action for topology level, due to the violated constraints "`Co4`" and "`Co7`".

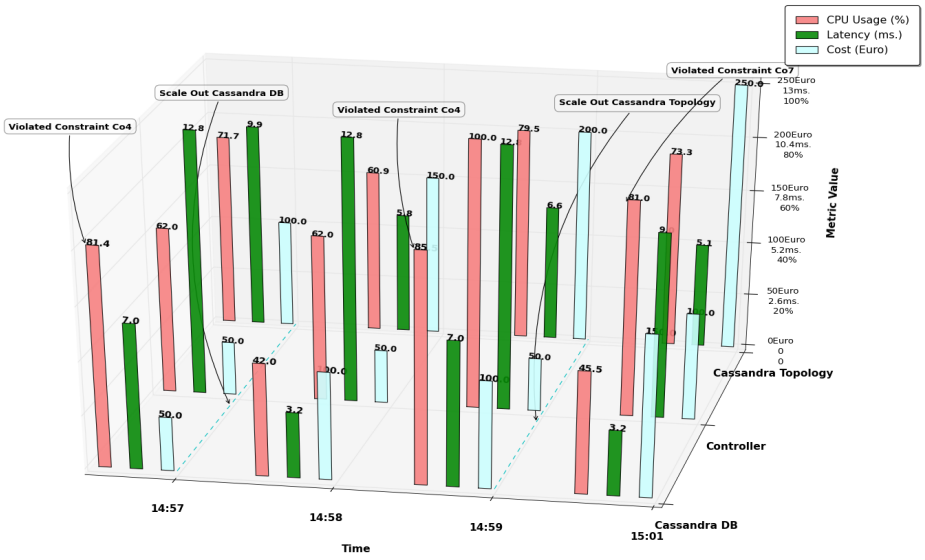

Fig. 6: Metrics (CPU usage, cost and latency) and elasticity actions for service units in `Data Service Topology`

6 Related Work

Yang et al. [1] support just-in-time scalability of resources based on profiles. Kazhamiakin et al. [9] consider KPI dependencies when adapting the service based applications. PRESS [10] and CloudScale [11] are examples of application resource elasticity frameworks which use prediction for reducing the number of over and under estimation errors to decrease the amount of SLO violations.

Malkowski et al. [3] support multi-level modeling and elastic control of resources for workflows in the cloud. Guinea et al. [12] develop a system for multi-level monitoring and adaptation of service-based systems by employing layer-specific techniques for adapting the system in a cross-layer manner. Elasticity control of storage based on resources and quality has been focused by different research works, e.g., adaptively deciding how many database nodes are needed depending on the monitored data in [2]. Wang et al. [13] propose an algorithm for software resource allocation considering the loads, analyzing the influence of software resources management on the applications performance. Kranas et al. [14] propose a framework for automatic scalability using a deployment graph as a base model for the application structure and introduce elasticity as a service cross-cutting different cloud stack layers (SaaS, PaaS, IaaS).

Compared to the above-mentioned work, we control elasticity not just in terms of resources but also in terms of quality and cost and use application structure for proposing an accurate multiple level control of elasticity of cloud services. Furthermore, they lack user-customized elasticity controls. We propose a user oriented elasticity control in which the user (cloud service creator, application developer, etc.) specifies how the cloud service should behave for achieving the elasticity property. Moreover, we argue in favor of an elastic services control aware of the structure of the elastic service, profiting from this knowledge for a multiple levels elasticity control of cloud services.

7 Conclusions and Future Work

We have presented an elasticity control system which enables multi-level specification of elasticity requirements and execution of automatic elasticity of cloud services by utilizing an abstract model for describing the structure of cloud services and allowing different elasticity metrics associated with the structure. Experiments show that our techniques bring several benefits for the user in utilizing resources while reducing the complexity of elasticity control covering not only resources but also cost and quality metrics.

Our system can be used by different roles existent in the cloud, from cloud providers to cloud consumers. With cross multi-level elasticity control capabilities, cloud providers could sell elasticity as a service (EaaS) to cloud consumers, allowing application code designers to specify elasticity in a high level manner and enforcing elasticity requirements for them while cloud consumers can deploy elastic services pre-packed with our techniques, which will automatically scale application components when needed. In our future work, further large-scale experiments will be conducted. Moreover, we will add a learning component for capturing actions effects on all metrics, and use other algorithms which take expected enforcement time into account for finding the best sequence of actions for fulfilling all the specified constraints.

References

1. Yang, J., Qiu, J., Li, Y.: A Profile-Based Approach to Just-in-Time Scalability for Cloud Applications. In: Proceedings of the 2009 IEEE International Conference on Cloud Computing. CLOUD '09, Washington, DC, USA, IEEE Computer Society (2009) 9–16
2. Tsoumakos, D., Konstantinou, I., Boumpouka, C., University), S.S.I., Koziris, N.: Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE Computer Society (2013) 34–41
3. Malkowski, S.J., Hedwig, M., Li, J., Pu, C., Neumann, D.: Automated control for elastic n-tier workloads based on empirical modeling. In: Proceedings of the 8th ACM international conference on Autonomic computing. ICAC '11, New York, NY, USA, ACM (2011) 131–140
4. Copil, G., Moldovan, D., Truong, H.L., Dustdar, S.: SYBL: an Extensible Language for Controlling Elasticity in Cloud Applications. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE Computer Society (2013) 112–119
5. IBM: IBM Cloud Computing Reference Architecture. https://www.opengroup.org/cloudcomputing/uploads/40/23840/CCRA.IBMSubmission.02282011.doc
6. OASIS Committee: Topology and Orchestration Specification for Cloud Applications (TOSCA), Working Draft. (2013)
7. Tai, S., Leitner, P., Dustdar, S.: Design by Units: Abstractions for Human and Compute Resources for Elastic Systems. IEEE Internet Computing **16**(4) (2012) 84–88
8. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing. SoCC '10, New York, NY, USA, ACM (2010) 143–154
9. Kazhamiakin, R., Wetzstein, B., Karastoyanova, D., Pistore, M., Leymann, F.: Adaptation of service-based applications based on process quality factor analysis. In: Proceedings of the 2009 international conference on Service-oriented computing. ICSOC/ServiceWave'09, Berlin, Heidelberg, Springer-Verlag (2009) 395–404
10. Gong, Z., Gu, X., Wilkes, J.: PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In: 2010 International Conference on Network and Service Management (CNSM). (oct. 2010) 9 –16
11. Shen, Z., Subbiah, S., Gu, X., Wilkes, J.: CloudScale: elastic resource scaling for multi-tenant cloud systems. In: Proceedings of the 2nd ACM Symposium on Cloud Computing. SOCC '11, New York, NY, USA, ACM (2011) 5:1–5:14
12. Guinea, S., Kecskemeti, G., Marconi, A., Wetzstein, B.: Multi-layered monitoring and adaptation. In: Proceedings of the 9th international conference on Service-Oriented Computing. ICSOC'11, Berlin, Heidelberg, Springer-Verlag (2011) 359–373
13. Wang, Q., Malkowski, S., Kanemasa, Y., Jayasinghe, D., Xiong, P., Pu, C., Kawaba, M., Harada, L.: The Impact of Soft Resource Allocation on n-Tier Application Scalability. In: Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International. (may 2011) 1034 –1045
14. Kranas, P., Anagnostopoulos, V., Menychtas, A., Varvarigou, T.: ElaaS: An Innovative Elasticity as a Service Framework for Dynamic Management across the Cloud Stack Layers. In: 2012 Sixth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS). (july 2012) 1042 –1049