Cloud Application Resource Mapping and Scaling Based on Monitoring of QoS Constraints

Xabriel J. Collazo-Mojica
S. Masoud Sadjadi
School of Computing and Information Sciences
Florida International University
Miami, FL, USA
{xcoll001, sadjadi}@cs.fiu.edu

Jorge Ejarque
Rosa M. Badia
Grid Computing and Clusters Group
Barcelona Supercomputing Center
Barcelona, Spain
{jorge.ejarque, rosa.m.badia}@bsc.es

Abstract—**Infrastructure as a Service (IaaS) clouds promise unlimited raw computing resources on-demand. However, the performance and granularity of these resources can vary widely between providers. Cloud computing users, such as Web developers, can benefit from a service which automatically maps performance non-functional requirements to these resources. We propose a SOA API, in which users provide a cloud application model and get back possible resource allocations in an IaaS provider. The solution emphasizes the assurance of quality of service (QoS) metrics embedded in the application model. An initial mapping is done based on heuristics, and then the application performance is monitored to provide scaling suggestions. Underneath the API, the solution is designed to accept different resource usage prediction models and can map QoS constraints to resources from various IaaS providers. To validate our approach, we report on a regression-based prediction model that produces mappings for a CPU-bound cloud application running on Amazon EC2 resources with an average relative error of 17.49%.**

Index Terms—**cloud computing; QoS; resource allocation.**

I. Introduction

Cloud computing presents the illusion of infinite capacity of computational resources. In the case of Infrastructure as a Service (IaaS) clouds, these resources are typically offered in bundles with specific amounts of CPU, Memory, and Network. Solution developers are thus presented with the problem of ensuring the performance non-functional requirements of an application by mapping it to one or more of these bundles, and to monitor and change this mapping if the workload changes. We present our work on an autonomic service which monitors Quality of Service (QoS) metrics of cloud applications and suggests bundle mappings which would ensure the required performance.

The problem of resource allocation in the cloud has been studied before, and various techniques to solve it has been proposed. Ganapathi et at. [1] utilize statistical machine learning to predict resource usage of an application in the cloud. Islam et al. [2] estimate CPU resource usage by simulating a cloud provider. Previous solutions have monitored low-level resources for their prediction, i.e., they would monitor CPU usage. Our solution targets QoS assurance by learning if the currently allocated resources are delivering the required QoS constraints. That is, we monitor if the application's performance non-functional requirements are being met (e.g. the response time), and based on this knowledge, we then adjust the resource allocation.

The key idea of our solution is the use of a Service Oriented Architecture (SOA) approach in which users provide a descriptive model of their application and get back mappings in various IaaS providers. These mappings emphasize the assurance of the Quality of Service metrics from the applications model. An initial mapping is done based on heuristics, and then we monitor the application's performance to provide scaling suggestions via a callback interface. Underneath this API, the solution accepts different resource usage prediction models and allows allocation in different IaaS providers.

The main technical challenges of this work were the experimentation with alternative regression techniques, and the implementation of a resource usage prediction engine. In this work, we propose the use of a linear regression model that provides adequate performance as well as good accuracy. For the implementation of the engine, we developed a rule-based system that calculates which IaaS resource bundles can ensure the applications QoS constraints. The solution then recommends the bundles with the best fit for various IaaS providers.

To validate our approach, we present three experiments. In the first experiment, we test for correct behavior with a simulation test that sends an application model and monitoring data to the prototype system. In our second experiment, we run the same model against a real IaaS environment. We gather the performance data and then scale the application according to the solution's suggestions. Finally, we also present a time and scalability analysis of the solution. The results show that the prototype correctly ensures QoS constraints of a CPU-bound cloud application.

The main contributions of this paper follow. First, we present the design of an autonomic solution for cloud application resource mapping and scaling based on monitoring of QoS constraints. Second, we provide details on a prototype implementation and how we dealt with the technical challenges. Finally, we asses the validity of the idea by presenting experiments on functionality and scalability.

II. Background and Methodology

A. Background

1) Previous Work: This work is part of our overarching "Distributed Ensemble of Virtual Appliances" (DEVA) project. In [3], we discussed the advantage of simplifying the solution development workflow by presenting the developer with a model-based visual designer to draft the software dependencies and resource requirements of their cloud application.

In [4], we formally defined our DEVA model approach. This model is based on the notion of Virtual Appliances, defined by Sapuntzakis et al. [5], that represent self-configurable software applications and OS as updatable image files. These appliances can be instantiated on top of IaaS providers such as Amazon EC2 [6]. DEVAs represent groups of Virtual Appliances with QoS constraints between each appliance, and general policies for the whole ensemble.

2) Problem Definition: Given that we have a model of an application's architecture and its desired QoS constraints, we can think of a model-to-instance transformation to different IaaS providers. For this transformation to work effectively, we also need a model representation of the performance of the application on top of resources from any given provider. That is, given a DEVA model, known IaaS providers and its bundles, and potentially known workloads, we want to transform the model (i.e. map the model) to instances that would ensure the QoS constraints specified in the model.

B. Methodology

1) Approach: To do this QoS-to-Resources mapping, we propose the steps illustrated in Figure 1. First, a user designs the DEVA model in the DEVA-Designer. When ready to deploy, the user chooses an IaaS provider. This request is (1) sent to a Transformation Engine, which has no previous knowledge of the submitted model, and thus delegates the creation of a preliminary mapping to an API Mapper (2). The Engine then makes the proper API calls to the specific IaaS provider (3). The IaaS provider (4, 5) instantiates the model. Note that the IaaS provider does not know about DEVAs, and only processes its own API calls. Note further that in this work we assume that the software provisioning is already done, i.e., that virtual appliances are available in the provider's image repository. Monitoring is done on each appliance to gather QoS data (6, 7). This data is eventually (8) fed back to the Transformation Engine. The Engine then (9) decides whether the currently assigned resources are appropriate for the QoS specifications in the model. If it is the case that a change of resource allocation is needed, the engine delegates the construction of a change request (10), and then sends that change to the IaaS provider (11). Finally, the provider makes the necessary changes to the instance to better comply with the model's QoS constraints (12).

2) Limitations: Our approach depends on previous monitoring data to make good resource mappings. To have good sample points, we propose the following. As we target cloud applications, the cloud developer (i.e. the user of our solution) first deploys their application in a "staging" mode. In this mode, the cloud developer runs performance tests on his application using different IaaS bundles. These performance tests generate data points that are sent to the proposed Monitoring API, and therefore improve the accuracy of our solution. As typical IaaS instances are billed by the hour, the cost of launching an application on different bundles for performance testing is negligible compared to the eventual QoS improvement. Although we are working in automatizing this learning phase, we do not report it as part of this paper.

III. Design and Implementation

In this section, we report on the main objectives driving the design of our solution. We also include implementation details on how we dealt with the technical challenges.

A. Design

We have designed the Transformation Engine as a service. In designing this service, we had two main Design Objectives:

1) to have a simple interface that can be used by our research team as well as others, and
2) to create a service that supports various resource mapping techniques for experimentation.

Although this work is integrated with the DEVA-portal project, it can also be used as a stand alone solution. This is why we chose to interact in terms of Resource Description Framework (RDF) [7] tuples. That is, the API expects and returns DEVA models wrapped in this interoperable model representation framework. Other research teams can use our mapping service by either using our DEVA modeling approach, or by first parsing their application representation to a DEVA using techniques such as OWL's Web Ontology Language [8]. This allows us to comply with Design Objective 1.

Table I presents the proposed REST API. We chose the REST technology because of Design Objective 1. First, a POST is done as explained in Section II-B1. This request has to include the DEVA model to be transformed. The API processes the model and responds with a link to the newly created resource. A subsequent GET to that link will return the transformed DEVA with the newly generated mappings. An API call to delete a mapping is also provided.

Monitoring data is expected as POST requests that include data points. The data points collected include the amount of the metric being monitored. A callback API subscribes with a POST or unsubscribes with a DELETE consumer that want to monitor model changes. Model mapping changes trigger these callbacks. Note that this design does not hint at any specific resource allocation technique, and thus allows us to implement a solution that complies with Design Objective 2.

B. Implementation

Underneath the API, the transformation engine is implemented as a simple resource usage prediction framework. For an initial mapping, the solution follows the steps presented in Figure 2. First, the DEVA model is received by the API. Then,

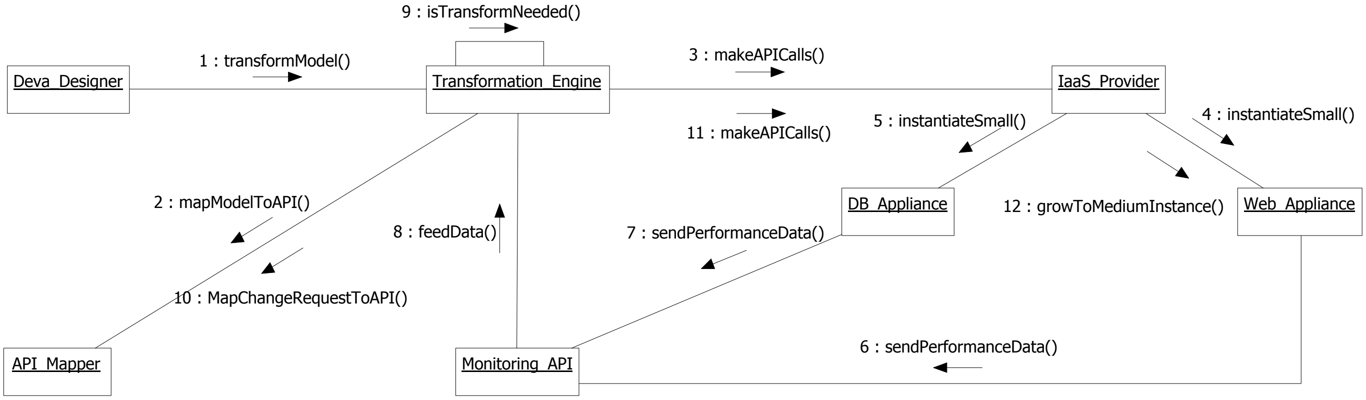


Fig. 1. UML Collaboration Diagram modeling the interaction of the solution with an IaaS provider.

REST HTTP API	Description
POST /deva/mappings	**Expects**: a DEVA model in RDF format in the body. **Optional**: a :callback_URL parameter if a callback_URL is provided, then it will be subscribed to this mapping. **Returns**: a HTTP 201 created status, with a link to the newly created mapping. Note that clients must persist the :uuid for later recall. **Error**: An HTTP 400 bad request if the model could not be parsed as a valid DEVA RDF.
GET /deva/mappings/:uuid	**Expects**: empty body **Returns**: an RDF/XML representation of the DEVA model with the mapping done to all supported IaaS providers. **Error**: a HTTP 404 not found if mapping does not exist.
DELETE /deva/mappings/:uuid	**Expects**: empty body **Returns**: a HTTP 200 OK. **Error**: a HTTP 404 not found if mapping does not exist.
POST /deva/mappings/:uuid/data_point	**Expects**: performance monitoring data of appliances of the specified mapping. Data Point should specify Appliance id, Connection id, and data value. (I.e. {appliance = 2, connection = {db-consumer, db-provider}, data_value = 500}) **Returns**: a HTTP 200 OK. **Error**: a HTTP 404 not found if mapping does not exist.
POST /deva/mappings/:uuid/subscriptions	**Expects**: an URL callback address to query when/if the specified mapping changes. Mappings could change in response to underprovisioned / overprovisioned resources as attested by monitoring data. The URL can be any valid endpoint. **Returns**: a HTTP 200 OK. **Error**: A HTTP 404 not found if mapping does not exist. **Error**: A HTTP 400 bad request if the callback_URL is not provided or malformed.
DELETE /deva/mappings/:uuid/subscriptions	**Expects**: A :callback_url parameter. **Returns**: A HTTP 200 OK. **Error**: a HTTP 404 not found if mapping does not exist. **Error**: A HTTP 400 bad request if the callback parameter is not provided or malformed.

TABLE I
APPLICATION PROGRAMMING INTERFACE FOR THE SOLUTION.

the model is processed by a set of rules that identify the virtual appliances included. For each one of the virtual appliances, a check is done on whether the engine has sufficient data to apply resource usage prediction based on machine learning techniques. If no sufficient data points have been gathered, then the solution applies a set of heuristic rules. If sufficient data is present, then it applies machine learning. In any case, a transformed model is created which includes IaaS bundles mappings. In the current prototype, we built an engine that utilizes rules to accomplish the above steps. Specifically, we are using the rules inference support that comes with the JENA 2.6.4 semantic web framework [9].

For the machine learning phase, we chose a multivariate linear regression model. We are experimenting with other machine learning techniques, but we only report on the regression-based one in this paper. The machine learning module receives the data points collected from the Monitoring API (see Figure 1) and estimates the QoS being achieved with the current resource allocation. In general, the parameters estimated are as follows:

$$\begin{aligned} targetMetric &= A_1 * CPU + A_2 * Memory + \\ &\quad A_3 * Network + A_4 * Workload + B \end{aligned}$$

Where the $targetMetric$ would be the metric we are trying to ensure. Each one of the A_i coefficients describe how important the estimation of the ith term is.

For example, given enough data points, a CPU-bound application will have a large A_1 coefficient compared to the others. The $Workload$ and the $targetMetric$ are known, but the CPU, $Memory$ and $Network$ parameters are not. This formula has three degrees of freedom, therefore trying to optimize the parameters would be expensive and many data points would be needed.

To avoid this expensive calculation, we apply a key insight: given that most IaaS providers offer bundles, rather than arbitrary combinations of CPU, $Memory$ and $Network$, we can reduce the complexity of the problem by estimation on bundled resources instead of trying to guess if an application

Amazon [6] ("Instance type")	**t1.micro**: 2.2Ghz (varies), 613MB, "Low" network **m1.small**: 1.1GHz, 1.7GB, "Moderate" network **m1.large**: 4.4GHz, 7.5GB, "High" network **m1.xlarge**: 8.8GHz, 15GB, "Moderate" network
Rackspace [11] ("Flavor")	**Flavor 1**: 256 MB RAM **Flavor 2**: 512 MB RAM **Flavor 3**: 1024 MB RAM **Flavor 4**: 2048 MB RAM **Flavor 5**: 4096 MB RAM **Flavor 6**: 8192 MB RAM **Flavor 7**: 15872 MB RAM
ElasticHost [10] (No bundles.)	Anything in the following ranges: 2-20Ghz CPU, 1-8GB RAM

TABLE II
EXAMPLE RESOURCE BUNDLES OF THREE IAAS PROVIDERS.

is bound by any specific resource:

$$targetMetric = A_1 * Bundle + A_2 * Workload + B \quad (1)$$

We further reduce the calculation and estimation errors by keeping a regression table that includes all available bundles. This effectively reduces the problem to a simple linear regression:

$$Bundle \longrightarrow targetMetric = A_1 * Load + B \quad (2)$$

Table II presents a sample of the resource bundles available from three different IaaS providers. For each provider, we construct a regression that matches each of their resource bundles against the application, and then we choose the best fit (I.e. the cheapest one). Some $targetMetric$s should be approached by the left, and others by the right, so we take this into account for the fit. For example, a "maximum response time" metric would be fit by the left, while a "minimum required throughput" metric would be fit by the right.

Note that some IaaS providers, such as ElasticHost [10], do not provide discrete bundles and instead the customer can choose exactly the amount of resources needed. In these cases, we simply quantize the range according to the bundles available from an arbitrary competitor.

IV. EXPERIMENTS AND ANALYSIS

The following experiments were designed as a proof of concept for our approach. For each one of them, we describe the experiment, present results and elaborate a short analysis.

A. *Experiment 1 - Simulation*

For this experiment, we test our Transformation Engine with a DEVA model composed of one Virtual Appliance that simulates the behavior of a CPU-bound Web Framework like Ruby on Rails. The model includes a maximum response time constraint. We first train the system with synthetic data points that cover various IaaS standard bundles as seen in Table II. After the training, we observe what mappings the system suggest to comply with different response time targets for a fixed workload of 3 request per second. Figure 3 shows the results. On the x-axis, we present response time targets, while on the y-axis we present the suggested mapping to the bundles of a particular IaaS provider. Note that this experiment is a validation of the approach and that the numbers do not reflect the real performance of the IaaS service from Amazon, Rackspace or ElasticHost.

Subfigure 3a presents the simulation results for mapping the model to the bundles of Amazon. As can be seen, for very strict response times of 100 to 400ms, a mapping to a "m1.xlarge" bundle is required. As the response time constraint relaxes, the solution maps the model to smaller resource bundles.

Subfigure 3b presents the simulation results for mapping the model to the bundles of Rackspace. Although the mapping was gradual on the case of Amazon, in this case we can observe that for a CPU-bound model and a target response time of around 600ms, the solution suggest that we switch from a "flavor 4" bundle to a "flavor 6", skipping over "flavor 5".

Subfigure 3c presents the simulation results for mapping the model to the quantized bundles of ElasticHost. According to the simulation, a mapping can be achieved for targets above 450ms, yet a response time of 425ms or less can not be achieved.

B. *Experiment 2 - Real Allocation*

For this experiment, we train the system in a similar way as in Experiment 1, but data is gathered from real instances on Amazon EC2 running a CPU-bound Ruby on Rails application. We provision instances for the standard Amazon bundles as seen in table II and then we generate a positive slope linear workload on each of the bundles to gather performance data points. We plot the mapping our solution recommends for workloads of 3, 6, and 10 requests per second. We also include two additional baseline mappings for comparison: a mapping that always over-provisions, and a mapping that always under-provisions resources.

Figure 4 presents the results. Note that the baseline mapping that always under-provisions is not able to fulfill the QoS, while the baseline that always over-provisions can fulfill the QoS, but utilizes a maximum of resources all the time. We can observe that our solution suggests bundle mappings that depend on the target metric, in this case the response time, as well as in the workload. Further, our solution fulfills the QoS of the application with a minimum relative error of 5.61%, maximum of 34.07%, and average of 17.49%.

C. *Experiment 3 - Time and Scalability Analysis*

For this experiment, we asses the time complexity and scalability of the solution. We ran these experiments on a 2.8GHz Intel i7 quad core machine with 8GB of available RAM. Figure 5 presents the results.

For the time analysis, we use the solution's API to POST a new model, POST a variable number of performance samples to the monitoring API, and then GET back the transformed models with mappings. Note that this is a worst case scenario, as normally the performance samples will not be gathered this fast. On subfigure 5a, we plot the response time of our solution against the number of collected data points from the monitoring API. The results suggest that the prototype can respond to GET model requests, which could trigger the

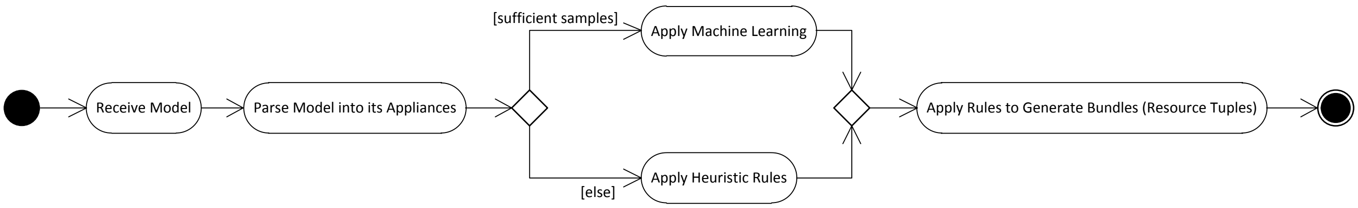
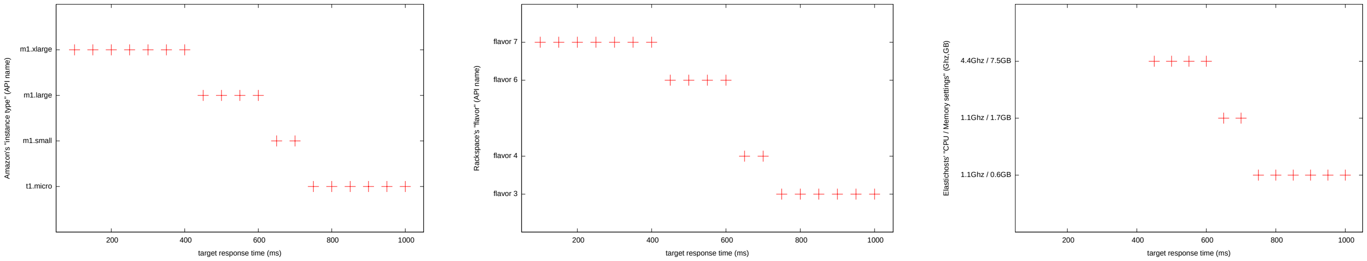


Fig. 2. UML Activity Diagram modeling the steps to map a DEVA model to IaaS Resources.



(a) Amazon Mapping.

(b) Rackspace Mapping.

(c) ElasticHost Mapping.

Fig. 3. Evaluation of the solution using simulated data points for a fixed workload of 3 request per second.

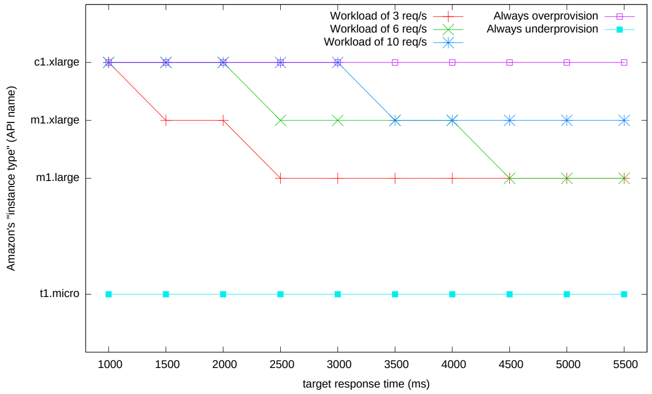


Fig. 4. Evaluation of the solution running a CPU-bound application on top of Amazon EC2. Note that results are discrete data points, but are shown with lines with points.

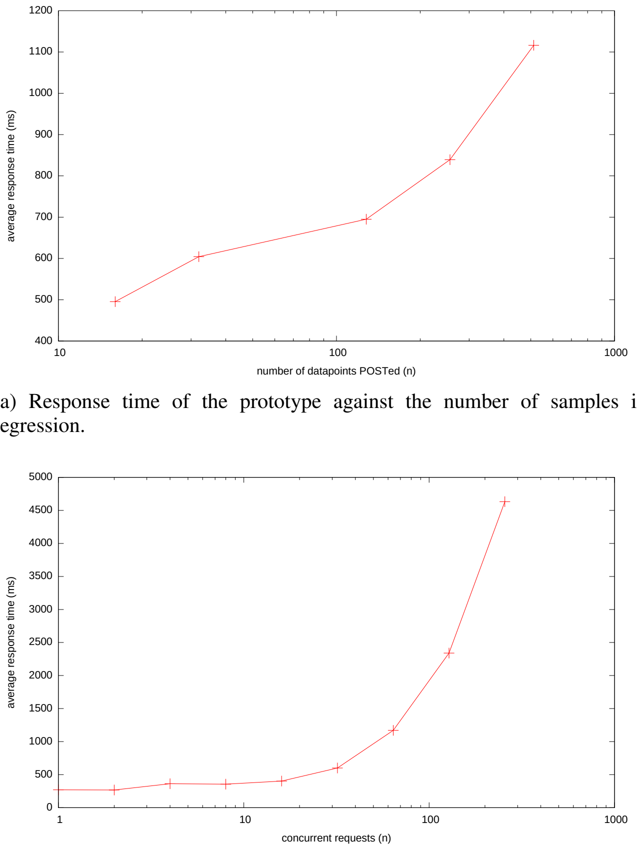
machine learning algorithm, in less than a second for up to 256 gathered data points. For a sample size of 512, we get a response time of 1116ms. For bigger sample sizes, the performance starts to quickly degrade in a quadratic manner. Nonetheless, experiential data suggest that we only need the performance test data points that we discussed in Subsection II-B2 and recent workload data points to achieve acceptable accuracy.

For the scalability analysis, we fix the acquired data points for a particular model to 64, and plot the response time of our solution against concurrent GET model requests. Subfigure 5b presents the results. The solution can achieve a subsecond response time for up to 32 concurrent requests. At 64 concurrent requests, the solution can respond in an average of 1170ms. For bigger concurrent requests, the performance starts to quickly degrade in a quadratic manner, even more so than in the previous experiment. For practical cases, our solution can respond to a maximum of 100 concurrent requests with an average response time of 2000ms.

V. Related Work

In [12], Stewart and Chen presented an implementation of an offline profile-driven performance model for cluster-based multi-component online services. Their model includes a detailed specification of the application. Although our solution also utilizes a detailed specification of the application, our work in [3] makes it straightforward for a solution developer to construct it. Additionally, we do online monitoring of the application to respond to workload changes. In [13], Sadjadi et al. proposed a regression model that estimates CPU usage for long-running scientific applications. In our work, we are estimating bundles of CPU, Memory, and Network, and the solution is targeting web cloud application worklaods.

In [2], Islam et al. estimate CPU usage by simulating a cloud provider. They contrast the use of linear regression and neural networks. Our work includes both a simulation as well as a real experiment of the solution on top of Amazon EC2, although we do not compare different machine learning models. In [14] Villegas and Sadjadi presented an IaaS solution that can have as input a model of a cloud application with non-functional specifications. Our work is complementary, as our DEVA models could be used for input to their solution. Also, in our work, we do not assume that the IaaS provider understands our model, and thus our solution is IaaS-agnostic. In [1], Ganapathi et al. utilize statistical machine learning to predict resource usage of an application in the cloud. Their workload is similar to the workload of [13], that is, batch processing of long-running jobs. We focus on web cloud application workloads.



(a) Response time of the prototype against the number of samples in regression.

(b) Response time of the prototype against concurrent requests with samples fixed to 64.

Fig. 5. Time and Scalability Analysis of the solution.

Some IaaS clouds like Amazon's EC2 already provide an auto-scaling API [6]. These APIs monitor low-level resources like CPU-usage, instead of our approach of directly monitoring key QoS metrics like response time. These vendor APIs focus on resource usage, while our solution's focus is on application performance non-functional requirements (i.e. QoS constraints).

In [15], Ejarque et al. propose the usage of semantics for enhancing the resource allocation in distributed platforms. They propose a set of extensions in resource ontologies and a set of rules for modeling resource allocation policies. A similar approach has been followed in their subsequent paper [16]. In these two cases, rules are used to model equivalences and mappings between the different cloud providers models. Thus, when the system receives a request following a providers model, it can be automatically transformed to another provider by applying the mapping rules to the original request. Our work is complementary, as we apply this rule mapping approach to a different problem. We map application descriptions, provided as DEVA models, into IaaS resource bundles.

VI. Conclusion

In this paper, we first presented the design of an autonomic solution for cloud application resource mapping and scaling based on monitoring of QoS constraints. We then provided details on the prototype implementation and how we dealt with the technical challenges. Finally, we assessed the validity of the approach by presenting experiments on functionality and scalability.

For future work, we intend to expand the prototype in two directions. In the near future, we will introduce other machine learning algorithms into our resource allocation framework and produce a comparison to see which techniques work best for this problem. In the longer term, we intent to expand our solution to consider not only vertical, but also horizontal scaling of Virtual Appliances. That is, to dynamically modify the DEVA model architecture if the QoS requirements are hard to achieve with the current one.

Acknowledgment

This work was supported in part by a GAANN Fellowship from the US Department of Education under P200A090061 and in part by the National Science Foundation under Grant No. OISE-0730065 and IIP-0829576.

References

[1] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," in *IEEE 26th International Conference on Data Engineering Workshops*, 2010, pp. 87–92.

[2] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," in *The International Journal of Grid Computing and Escience*. Natl ICT Australia, Software Engn Res Grp, Sydney, NSW, Australia, 2012, pp. 155–162.

[3] X. J. Collazo-Mojica, S. M. Sadjadi, F. Kon, and D. D. Silva, "Virtual environments: Easy modeling of interdependent virtual appliances in the cloud," *SPLASH 2010 Workshop on Flexible Modeling Tools*, Aug 2010.

[4] X. J. Collazo-Mojica and S. M. Sadjadi, "A Metamodel for Distributed Ensembles of Virtual Appliances," in *International Conference on Software Engineering and Knowledge Engineering*, May 2011, pp. 560–565.

[5] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum, "Virtual appliances for deploying and maintaining software," *USENIX Large Installation Systems Administration Conference*, pp. 181–194, Aug 2003.

[6] "Amazon Elastic Compute Cloud," March 2012. [Online]. Available: http://aws.amazon.com/ec2/

[7] "Resource Description Framework (RDF)," March 2012. [Online]. Available: http://www.w3.org/RDF/

[8] "OWL Web Ontology Language," March 2012. [Online]. Available: http://www.w3.org/TR/owl-ref/

[9] "Apache JENA," March 2012. [Online]. Available: http://incubator.apache.org/jena/

[10] "ElasticHosts," March 2012. [Online]. Available: http://www.elastichosts.com/

[11] "Rackspace Cloud," March 2012. [Online]. Available: http://www.rackspace.com/cloud/

[12] C. Stewart and K. Shen, "Performance Modeling and System Management for Multi-component Online Services," in *2nd Symposium on Networked Systems Design & Implementation*, May 2005, pp. 71–84.

[13] S. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. Collazo-Mojica, "A modeling approach for estimating execution time of long-running scientific applications," in *IPDPS*, 2008, pp. 1–8.

[14] D. Villegas and S. M. Sadjadi, "Mapping Non-Functional Requirements to Cloud Applications," *International Conference on Software Engineering and Knowledge Engineering*, Jun. 2011.

[15] J. Ejarque, R. Sirvent, and R. Badia, "A Multi-agent Approach for Semantic Resource Allocation," in *Cloud Computing Technology and Science*, 2010, pp. 335–342.

[16] J. Ejarque, J. Alvarez, R. Sirvent, and R. Badia, "A Rule-based Approach for Infrastructure Providers' Interoperability," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, 2011, pp. 272–279.