Dynamic Deployment and Auto-scaling Enterprise Applications on the Heterogeneous Cloud

Satish Narayana Srirama, Tverezovskyi Iurii, Jaagup Viil
Institute of Computer Science, University of Tartu
J. Liivi 2, Tartu, Estonia
srirama@ut.ee

***Abstract*—Over the past years, organizations have been moving their enterprise applications to the cloud with the aim of reducing infrastructure ownership and maintenance costs, and to take advantage of the elasticity and heterogeneity of the cloud. This paper joined the approaches of multi-cloud deployment using CloudML and identifying the ideal resource provisioning and deployment configuration using an optimization model, in order to dynamically scale an enterprise application across multiple clouds, without any user intervention. The approaches are discussed in detail along with the introduced extensions. Benchmark experiments were conducted on Amazon cloud infrastructure, based on one system with a single scalable component and two other systems with the basic workflow control structures, parallel and exclusive. The results of the experiments suggest that the approach is plausible for dynamic deployment and auto-scaling any web/services based enterprise workflow/application on the cloud.**

***Index Terms*—Cloud computing; dynamic deployment; auto-scaling; enterprise applications; optimal resource provisioning;**

I. Introduction

Cloud computing [1] has gained significant popularity over past few years. Employing service-oriented architecture (SOA) and resource virtualization technology, public clouds provide the highest level of scalability for enterprise applications with highly dynamic load. While, cloud, with its intrinsic features such as elasticity and utility computing is interesting, for migrating enterprise applications to the cloud, one should worry about the ideal deployment configuration of the applications and auto-scaling them on the cloud.

It is obvious that all major players in cloud solutions industry have their own tools and frameworks designed to make deployment of applications in the cloud easier. For example; Amazon has a number of solutions for dynamic deployment of applications such as AWS Elastic Beanstalk and AWS CloudFormation [2]. Similarly, Google Cloud Deployment Manager [3], allows to declare, deploy and manage enterprise applications using the concept of templates. However, since different cloud providers have different deployment mechanisms, it is difficult to span a system across multiple clouds.

Several projects have studied the interoperability across multiple clouds and as part of EU FP7 REMICS project [4] we have developed the CloudML [5], [6], a domain specific language, using which one can model the provisioning and deployment of multi-cloud systems. We also have developed a models@run-time execution engine which ensures the enactment of the specified model on the respective clouds. CloudML is actively being developed further in projects such as PaaSage[1], CloudScale[2] and ModaClouds[3].

Once an application is migrated, it should be auto-scaled properly on the cloud to take advantage of cloud's elasticity. In N-tier enterprise applications or workflows, generally, different components will have different scaling requirements and finding an ideal configuration and having it to scale up and down, dynamically, based on the incoming requests is a difficult task. To be precise, since each deployment component or web service of an enterprise application, or task of a workflow requires different processing power to perform its operation, at the time of load variation it must scale in a manner fulfilling its specific requirements the most.

A number of auto-scaling policies have been proposed so far. Some of these methods try to predict next incoming loads, while others tend to react to the incoming load at its arrival time and change the resource setup based on the real load rate rather than predicted one [7], [8]. However, in both methods there is a need for an optimal resource provisioning policy that determines how many servers of a particular type (taking advantage of the cloud heterogeneity) must be added to or removed from the system, in order to fulfill the load, while minimizing the cost.

Another key challenge here arises from the fact that cloud providers generally charge the resource usage for fixed time periods (e.g. hourly). Current methods of auto-scaling take into account a range of related parameters such as, incoming workload, CPU usage of servers, network bandwidth, response time, processing power and cost of the servers [9], [10]. However, none of them incorporates the life duration of a running server and current deployment configuration.

To address the above challenges in auto-scaling enterprise applications, we designed and developed an ILP (Integer Linear Programming) model that takes into account all major factors involved in scaling including periodic cost, configuration cost and processing power of each instance type, instance count limit of clouds, and life duration of each instance with customizable level of precision, and outputs an optimal combination of possible instance types suiting each component of an enterprise application the most [11].

[1] http://www.paasage.eu/
[2] http://www.cloudscale-project.eu/
[3] http://www.modaclouds.eu/

This paper tried to join both the approaches of multi-cloud deployment using CloudML and identifying the ideal resource provisioning and deployment configuration using the optimization model, in order to dynamically scale an enterprise application across multiple clouds, without any user intervention. Extensions were introduced to CloudML features to incorporate the scale up components and load balancers, and a framework was built to identify the changes in the model automatically, and ensure the changes as and when they appeared, using the models@run-time engine.

We created a simulation tool based on the proposed model and conducted real-time benchmark experiments on Amazon cloud. The benchmarks are based on one system with a single scalable component and two other systems based on Parallel (AND) and Exclusive (XOR) workflow control structures. The results of the experiments suggest that the approach is plausible for dynamic deployment and auto-scaling any web/services based enterprise workflow/application on the cloud.

The paper is organized as follows: Section II discusses the dynamic deployment of multi-cloud systems with CloudML. Section III summarizes the optimization model. Section IV discusses the combination of the approaches and Section V provides a detailed evaluation of the combined approach. Section VI later discusses the related work, while section VII concludes the paper with future research directions.

II. Dynamic deployment of multi-cloud systems with CloudML

In recent years, there has emerged dynamically adaptive systems (DAS) that will ease the development, adaption and continuous design of complex software systems. Usually, these systems are focused on the application and not the core platform and infrastructure. Due to the lack of software engineering approaches and methods widely accepted across multiple cloud computing providers, a model-based framework called CloudMF [12], was developed and is being managed as part of several European Commission projects.

CloudMF primarily enables multi cloud-based DAS and is made up of two primary components: 1) cloud modelling language, CloudML and 2) models@run-time environment. The first one is used to describe and create the model of the system. This includes information about the instances, set-up, deployment, etc., while the models@run-time is used to execute the provisioning itself and is the main architecture, which reflects the changes in the running system to the current model associated to it.

CloudML's main aim is to provide the users with a framework, which could be run on multiple cloud platforms and infrastructures, so that the user is free to migrate their applications to other vendors, as and when needed. To be able to describe the model of the system, CloudML allows the user to do it using JavaScript Object Notation (JSON), Ecore/XMI file or Java API.

Listing 1 shows how different nodes and artefacts are defined using CloudML. Nodes are self-explanatory, with information such as OS, memory and CPU requirements etc. Artefacts are used for defining dependencies on the instances and to install the required components, such as scripts, binaries, etc. and execute them, while bindings define how artefacts are dependent on each other. Several systems have been deployed with CloudML and the engine was shown to be efficient [6].

Listing 1: Example snippet of a CloudML JSON file

```
"nodeTypes" : [ {
        "id" : "myApp",
        "os" : "Debian",
        "provider": "aws-ec2",
        "compute" : [2, 4],
        "memory" : [1024, 2048],
        "storage" : [20480],
        "location" : "us-east-1",
        "sshKey" : "sshkey",
        "securityGroup" : "ports_open",
        "privateKey" : "KEY",
        } ],

"artefactTypes" : [ {
        "id": "mediaWiki",
        "retrieval": "wget http://../m_wiki.sh",
        "deployment": "sudo m_wiki.sh",
        "requires": [ {
                "id": "MySQLCapability"
                } ]
        } ]
```

III. Optimal resource provisioning for auto-scaling enterprise applications

While CloudML addresses the deployment issues, designing an auto-scaling model for entire enterprise application/workflow is a complex task. In these applications, generally, each component will have its own specific requirements to scale. Moreover, in every public cloud there are a range of instance types, each one having a different power/price rate. Whereas a *large* instance might be more beneficial for one task, a *medium* might have a better performance for another one, so we shall allocate different instance types to different tasks in order to reduce the total cost. Sometimes the optimal allocation for a task can be a composition of multiple types of instances. In addition, due to SLA (service-level agreement) of applications, some of the tasks may even have to run on completely different clouds, to support customers from different regions.

We studied the problem in detail and came up with a LP model based solution [11]. Input to the model includes incoming workload of each task, processing power, periodic cost and configuration time of instance types, maximum instance count limit of the cloud, and age of each running instance. The model provides the optimal number of instances from each type that must be added to or removed from cluster of each task, resulting in handling the workload and minimizing the cost. In summary, we created the notion of *time bag*s, by dividing an instance's paid life period (e.g. hour in case of Amazon EC2) into equal time periods. Thus, all the instances in the enterprise system, at any given time, will be in one of these time bags. This provides us a means to calculate the value of a running instance to the complete system, considering its age. The enterprise application is divided into component/tasks, each of them, running in a specific region.

The intuition for considering different parameters, the performance efficiency of the model, its implementation and comparison with solutions such as Amazon AutoScale, are

thoroughly addressed in [11]. The model is summarized here to make the current writing a consolidated one.

A. Description of the optimization model

Parameters of the model are listed below:
-$C_{r,t}$: Cost of a time period of instance type t running in region r.
-$CTB_{r,t}$: Cost of a time bag from instance type t running in region r. This cost is calculated by dividing the cost of a period of instance type t by total number of time bags.
-$CT_{r,t}$: Configuration time of instance type t running in region r. This value must be specified by time bag metric. For example in our experiments, we considered first 3 time bags as configuration time, a number approximated by the time taken by instances from Amazon EC2 to start up and customize the configuration.
-$KC_{r,t,tb}$: Killing Cost of time bag tb from instance type t running in region r.
-$RC_{r,t,tb}$: Retaining Cost of time bag tb from instance type t running in region r.
-$X_{r,t,tb}$: The number of instances in time bag tb from instance type t running in region r.
-$P_{r,t}$: Processing power of instance type t running in region r.
-$CCT_{r,t}$: Capacity constraint (or instance count limit) of instance type t running in region r.
-W_r: Workload of region r. This is the current incoming workload to the system and must be provided by the same metric as $P_{r,t}$. meaning that if $P_{r,t}$ is calculated by requests per second, $W_{r,t}$ must be provided as requests per second too.
-CC_r: Capacity constraint (or instance count limit) of region r. The number varies per cloud, e.g. a private cloud with limited capacity. Even in Amazon, by default, customer can launch only up to 20 instances.

The model has 2 *variables*:
-$N_{r,t}$: The number of new instances from instance type t running in region r that must be added to the system.
-$S_{r,t,tb}$: The number of instances of time bag tb from instance type t in region r that must be shut down.

The objective function of the model is as follows:

$$\begin{aligned} Min\ (&\sum_{i=1}^{n}\sum_{j=1}^{m}(N_{r_i,t_j} * C_{r_i,t_j} + N_{r_i,t_j} * (CT_{r_i,t_j} * CTB_{r_i,t_j})) \\ &+ \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{q} S_{r_i,t_j,tb_k} * KC_{r_i,t_j,tb_k} + \\ &\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{q}(X_{r_i,t_j,tb_k} - S_{r_i,t_j,tb_k}) * RC_{r_i,t_j,tb_k}) \end{aligned} \tag{1}$$

The objective function comprises sum of all costs attached to changing the arrangement of resources at any point of time. The cost function, sums cost of new instances and their configuration, killing cost of each instance that must be shut down and retaining cost of each instance that will continue living. For each region the model outputs the number of new instances of each instance type that must be added and number of instances of each time bag from each instance type that must be terminated so that the cost becomes minimal and all these following constraints are fulfilled.
-The workload constraint $\forall$ regions r $\in$ R:

$$\sum_{j=1}^{m}(N_{r,t_j} + (\sum_{k=1}^{q} X_{r,t_j,tb_k} - S_{r,t_j,tb_k})) * P_{r,t_j} \geq W_r \tag{2}$$

-The cloud capacity constraint $\forall$ regions r $\in$ R:

$$\sum_{j=1}^{m}(N_{r,t_j} + (\sum_{k=1}^{q} X_{r,t_j,tb_k} - S_{r,t_j,tb_k})) \leq CC_r \tag{3}$$

-Instance type capacity constraint $\forall$ instance types t $\in$ T_r:

$$N_{t_r} + (\sum_{k=1}^{q} X_{t_r,tb_k} - S_{t_r,tb_k}) \leq CCT_{t_r} \tag{4}$$

-Shutdown constraint $\forall$ time bags tb $\in$ $TB_{r,t}$:

$$S_{tb_{r,t}} \leq X_{tb_{r,t}} \tag{5}$$

-And:

$$N_{r,t} \geq 0, S_{r,t} \geq 0 \tag{6}$$

Killing cost and retaining cost actually specify how valuable a running instance is still for us. The more the instance lives in its current time period, the retaining cost becomes higher and the killing cost become lower, and thus the instance becomes less valuable. This guarantees that during scale-down the instances from the last time bags will have higher chance to be terminated. However, adding a new instance is bound to a new configuration process that triggers redundant cost which might make the addition of the new instance unprofitable. This is avoided by adding configuration cost to objective function.

The constraint (2) is defined to ensure that the new setup will fulfill the incoming workload in each region. Furthermore, the total number of instances in each region must not exceed its capacity; this is fulfilled using the constraint (3). The constraint (4) checks that the number of instances of each instance type in each region does not surpass its limit. Finally, constraint (5) makes sure that from each time bag the model does not shut down more instances than it contains.

The model is implemented in OptimJ [13] and using one of its free solvers, GLPK (GNU Linear Programming Kit) [14]. The cost and efficiency of the model are shown to be comparable to Amazon AutoScale in [11].

IV. Joining Auto-Scaling & Dynamic Deployment

To introduce our optimization model for load balancing into the CloudML, we added following new components.

- LoadBalancerEntity - A wrapper containing several CloudML internal components such as ComponentInstance, VMInstance, etc. This added component with the help of DNS rerouting (explained later), will allow the load to be redirected between multiple instances.
- ScalableComponent - This is an entirely new component added to the CloudML core, which allows the user to create flexible and scalable components in their model.

Scalable component can instantiate new virtual machines or a LoadBalancerEntity. During a scale up, the component will automatically add a load balancer to the scalable component and redirects the incoming traffic between available instances.

- DirectCommand - A component that will allow to run commands in the virtual machine after the execution and it can be added to the model at any given time.

In order to utilize the full prospects of scalability, we made every scalable component to have the ability to scale separately. To support this we added two algorithms: scaling up and scaling down. For practical use they were combined to a single module, that will take a configuration, which is a simple array and is produced using our LP model, and determines which instances to kill or create. For example, if the incoming, configuration is [1,0,2] and the instance types defined for the scalable component are [M1.micro, M1.small, M3.medium], the ideal deployment at that particular moment would be one M1.micro and two M3.medium instances. Also, to accommodate the traffic rerouting, we used a Domain Name System (DNS) caching resolver called DNS Unbound [15]. This allows us to redirect every communication through the load balancer, when the component is scaled up. DNS Unbound adds one additional step to the operating system before checking the DNS servers for an IP address - it checks whether the Unbound configuration file has a rule for that specific DNS name and if there is a match, the rule will be used.

The extended version of the CloudML and the optimization model are combined into a simple RESTful web-service application using Spring, which allows us to deploy any model and start the auto-scaling of the system with ease. For example, when submitting the initial JSON of the model to the URL */initial*, the model will be deployed to the specified platform, and when visiting */initial/start*, the automatic scaling of the system will start.

V. Evaluation of the Approach

We conducted three experiments to show that the proposed solution can be used for the dynamic deployment and auto-scaling a vast range of applications in the cloud. First experiment is similar to usual enterprise applications such as WikiPedia. The set up includes one scalable component, the application server, which has connection to an external database. The next two experiments were designed to show the parallel and exclusive workflow scalability with our approach.

A. Setup

For the test environment, we chose Amazon EC2, the most popular of cloud platforms. All instances were running Ubuntu 14.04 OS. We used three types of EC2 instances for all tests - M1.small, M3.medium and M3.large.

To simulate load in all tests, we used Tsung [16], an open-source multi-protocol distributed load testing tool, which allows to simulate load with various rates. Tsung later provides statistics such as system average response time, total number of requests and CPU load. To make the tests as realistic as possible, an archive of HTTP requests to ClarkNet's WWW server was used and the number of requests per second are scaled to the ranges (∼600), using Python scripts.

Considering that our LP model heavily relies on the maximum number of concurrent requests per second supported by each instance type ($P_{r,t}$), we measured this parameter as accurately as possible, by restricting the average CPU usage to ∼80% and the average response time to be under 500ms.

B. Scaling of a single component application

In this experiment we used MediaWiki [17], an open source Wikipedia replica, as a test application for the deployment. The database was populated with logs from publicly available WikiPedia database files. For each request, we requested a random page from MediaWiki, to simulate a real workload.

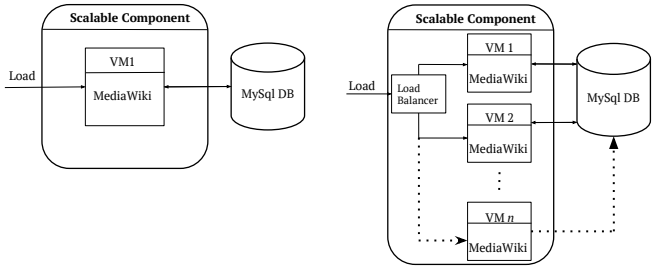

Fig. 1: The setup of the system before and after scale up

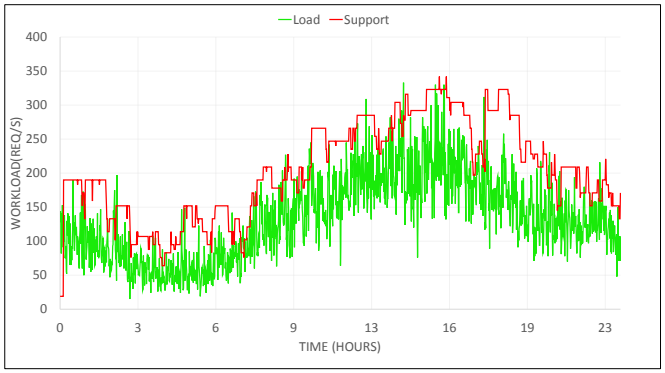

Fig. 2: Workload and system capacity during single scalable component experiment

After bombarding the scalable component with requests generated by Tsung, the modified configuration scaled up properly. Figure 2 shows the system's capacity and the workload for 24 hours. Figure 3 shows the number of different types of instances provisioned to the scalable component (M3.large was highly preferred). The experiment's results are summarized in table I, showing that the load was handled well.

C. Scaling of parallel and exclusive workflows

The goal of the second and third experiments is to show the possibility of building applications with multiple scaling components, that use parallel (AND) or exclusive (XOR)

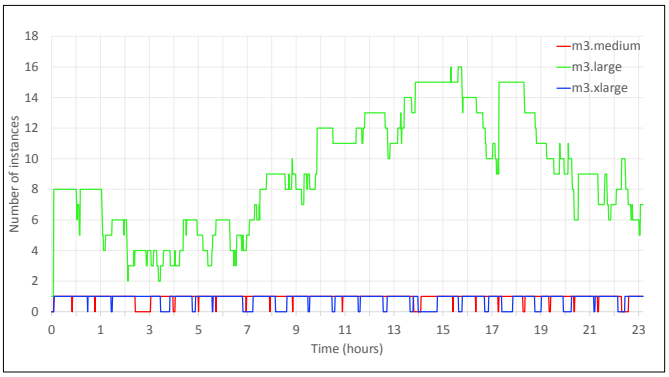

Fig. 3: Number of instances of each type during single scalable component experiment

Total number of requests	11,417,146
Average response time (sec)	0.26
Requests lost	201,036 (1.7%)
Successful requests	11,216,110 (98.3%)

TABLE I: Request success and failure count.

workflows. For these tests we used the system with three scalable components. In a parallel workflow, after component 1, components 2 and 3 are executed simultaneously, as opposed to exclusive workflows, where one of the components is executed based on specified constraints.

For the experiments, we chose to build a custom application to simplify deployment configuration and have more control over the system. The application consists of three different elements, that were developed using JavaScript and Express framework to support the requested model and for the server, we used NGINX. In order to consume a lot of CPU and memory, so that we can test the scale up easily, the application calculates Fibonacci numbers and returns them to the client.

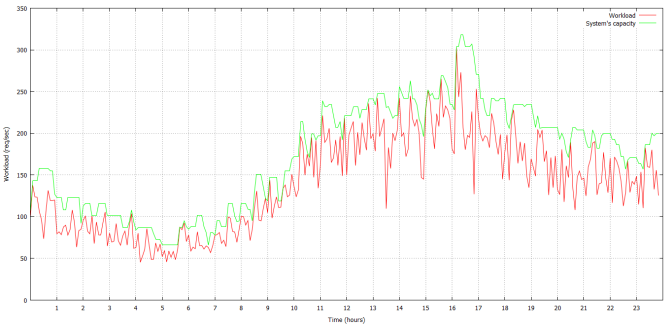

Fig. 4: Workload and system capacity during parallel workflow experiment (second component)

Figure 4 shows the system's capacity and Figure 5 shows the combination of instance types during the execution of the parallel workflow experiment, for the second component. From the results we observed that the system was able to provide the needed performance over 96% for the first component, which for the second and third component was 98%. We also

Fig. 5: No of instances of each type during parallel workflow experiment (second component)

calculated how slow the system became as the users increased. Average response time for M1.small instance in the initial state was 0.49 seconds and after the experiment, the average response time increased to ≈0.54 seconds.

For the third experiment, which considers exclusive workflows, the custom application had to be changed a bit - instead of simultaneously running the Fibonacci computations on all nodes, it was distributed between two components using a round-robin scheme.

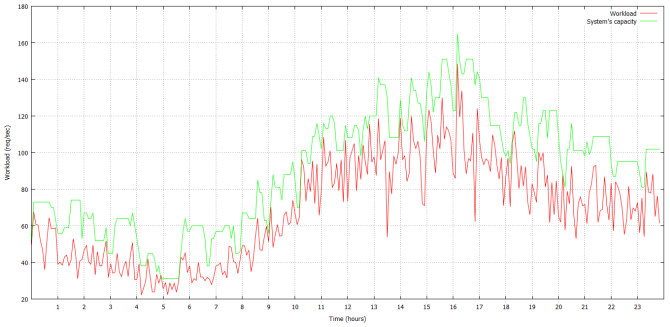

Fig. 6: Workload and system capacity during exclusive workflow experiment (third component)

Similar to the previous experiment, we noticed that the system was able to support the incoming workload in 96% for the first component and about 99% for the second and third component. Figure 6 shows the system's capacity during the execution of the exclusive workflow experiment, for the third component.

VI. Related work

Deployment and monitoring of distributed systems have been extensively studied over the past few years. Server management solutions, such as IBM Tivoli [18] can produce consistent and reproducible deployment configuration, even though not specifically targeted at cloud. Libraries such as jclouds[4], Simple Cloud[5], or DeltaCloud[6] have recently

[4]http://www.jclouds.org
[5]http://simplecloud.org/
[6]http://deltacloud.apache.org/

emerged to deploy and maintain cloud-based systems, however, they remain code-level tools. Similarly, frameworks such as Cloudify[7], Chef[8] and Puppet[9] provide language-dependent solutions, with capabilities for the automatic provisioning, deployment and monitoring of cloud based systems.

Regarding auto-scaling strategies, having been used by many auto-scaling services such as Amazon AutoScale [9], RightScale [10], threshold-based policies are very popular among users due to their simplicity. However, these methods require expert knowledge of load and cloud computing to set up an optimal service.

Regarding optimal resource provisioning policies, apart from LP several other technologies can also be used. Dutreilh et al. [19] used Reinforcement Learning and tried to improve the approach with better initialization and faster convergence to optimal policy. Urgaonkar et al. [20], using Queuing theory, experimented a network of queues in a multi-tier application. Harold et al. [21], using Control Theory, proposed a simple controller that produces the output based on average CPU usage. Xu et al. [22] utilized a fuzzy model to estimate CPU capacity needed for handling the incoming workload. We also have tried other models and in [8], we combined heuristics and queuing models with a reactive model for auto-scaling MediaWiki application.

LP was also used in other works such as by Mao et al. [23], where they proposed an LP model to find optimal combination of different instance types to handle the workload comprising of multiple class jobs within a variable deadline, assuming sequential running of jobs on an instance. In contrary, in our model we maximize resource usage with assigning a specific class of jobs to a cluster of multi-type instances.

VII. Conclusions and Future Work

The paper showed how we can join multi-cloud deployment using CloudML and identifying the ideal resource provisioning and deployment configuration using an optimization model, in order to auto-scale an enterprise application across multiple clouds. Currently, we are interested in adapting/remodelling enterprise applications for cloud migration. We propose remodelling and scheduling the applications, in a way that increases the intra-instance communication while reducing inter-instance communication, so that the applications will fit nicely to the cloud networks. The applications can be monitored for their performance and later partitioned with graph partitioning approaches. We studied the approach in migrating scientific workflows to the cloud [24]. Following the approach and joining the CloudML and optimal resource provisioning policy, we would like to achieve a framework to which any enterprise application can be provided, which would be studied, remodelled, migrated to, performance monitored and auto-scaled on the cloud, seamlessly.

[7]http://www.cloudifysource.org/

[8]http://www.opscode.com/chef/

[9]https://puppetlabs.com/

Acknowledgment

This research is supported by the Estonian Science Foundation grant PUT360. The authors would like to thank Alireza Ostovar for his contribution to the optimization model.

References

[1] M. Armbrust, A. Fox, R. Griffith *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] *Amazon Web Services (AWS) - Cloud Computing Services*. [Online]. Available: https://aws.amazon.com/

[3] *Cloud Deployment Manager Google Cloud Platform*. [Online]. Available: https://cloud.google.com/deployment-manager/overview

[4] REMICS, "Reuse and migration of legacy applications to interoperable cloud services." [Online]. Available: http://www.remics.eu/

[5] G. Goncalves, P. Endo, M. Santos, D. Sadok, J. Kelner, B. Melander, and J.-E. Mangs, "Cloudml: An integrated language for resource, service and request description for d-clouds," in *Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, Nov 2011, pp. 399–406.

[6] A. Sadovykh, A. Abhervé, S. Srirama, P. Jakovits, M. Smialek, W. Nowakowski, N. Ferry, and B. Morin, "Deliverable D4. 5 REMICS Migrate Principles and Methods," 2010.

[7] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09-12*, 2012.

[8] M. Vasar, S. N. Srirama, and M. Dumas, "Framework for monitoring and testing web application scalability on the cloud," in *Nordic Symp. on Cloud Computing & Internet Technologies (NORDICLOUD)*. ACM, 2012, pp. 53–60.

[9] *Amazon Auto Scaling*. [Online]. Available: http://aws.amazon.com/autoscaling/

[10] *Rightscale*. [Online]. Available: http://support.rightscale.com/

[11] S. N. Srirama and A. Ostovar, "Optimal resource provisioning for scaling enterprise applications on the cloud," in *The 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2014)*, 2014, pp. 262–271.

[12] N. Ferry, F. Chauvel, A. Rossini, B. Morin, and A. Solberg, "Managing multi-cloud systems with cloudmf," in *Proceedings of the Second Nordic Symposium on Cloud Computing and Internet Technologies*, ser. NordiCloud '13. ACM, 2013, pp. 38–45.

[13] *OptimJ*. [Online]. Available: http://www.ateji.com/optimj/index.html

[14] *GLPK optimizer*. [Online]. Available: http://www.gnu.org/software/glpk/

[15] DNS Unbound, "Unbound, a validating, recursive, and caching dns resolver." [Online]. Available: https://www.unbound.net/

[16] Tsung, "Open-source multi-protocol distributed load testing tool." [Online]. Available: http://tsung.erlang-projects.org/

[17] MediaWiki, "Open-source wiki package." [Online]. Available: https://www.mediawiki.org/wiki/MediaWiki

[18] T. Delaet, W. Joosen, and B. Van Brabant, "A survey of system configuration tools." in *LISA*, 2010.

[19] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant *et al.*, "Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow," in *7th Intl Conf. on Autonomic and Autonomous Systems (ICAS 2011)*, 2011, pp. 67–74.

[20] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," in *ACM SIGMETRICS Performance Evaluation Review*. ACM, 2005, pp. 291–302.

[21] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: challenges and opportunities," in *1st workshop on Automated control for datacenters and clouds*. ACM, 2009, pp. 13–18.

[22] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "On the use of fuzzy modeling in virtualized data center management," in *Int. Conf. on Autonomic Computing (ICAC'07)*. IEEE, 2007, pp. 25–25.

[23] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*. IEEE, 2010, pp. 41–48.

[24] S. N. Srirama and J. Viil, "Migrating Scientific Workflows to the Cloud: Through Graph-partitioning, Scheduling and Peer-to-Peer Data Sharing," in *Int. Conf. on High Performance Computing and Communications (HPCC 2014) Workshops*. IEEE, 2014, pp. 1137–1144.