Modeling the use of spot instances for cost reduction in cloud computing adoption using a Petri net framework

Maristella Ribas[1], C. G. Furtado[2], Giovanni Barroso[3]
[1]Techne Engenharia e Sistemas, São Paulo, Brazil
[2]Federal Institute of Ceará (IFCE), Fortaleza, Brazil
mari@techne.com.br, cjunior@ifce.edu.br,
gcb@fisica.ufc.br

Alberto S. Lima[3], Neuman Souza[3], Antão Moura[4]
[3]Federal University of Ceará, Fortaleza, Brazil
[4]Federal University of Campina Grande (UFCG), Brazil
{albertosampaio, neuman}@ufc.br, antao@dsc.ufcg.edu.br

Abstract
An effective decision-making about using Infrastructure as a Service (IaaS) resources in cloud computing projects is still a challenge to managers. We need to optimize resources use in cloud services, to obtain financial success in cloud projects. In this work, we propose a petri net framework to model possible cost savings using public clouds spot instances pricing scheme. The results from initial simulations indicate that spot instances can be a very interesting option for savings in autoscaling process.
Keywords—*Cloud computing, Spot Instances, BDIM, Petri nets..*

I. Introduction

Cloud computing is defined by NIST [1] as a model for enabling on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction. Usually, cloud services are categorized as Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

In a previous work [2], we proposed a novel framework that combined several of the most relevant factors (according to the current literature) to assist decision-makers in the evaluation of both SaaS and on-premises options choice. There, we identified that cloud services managers need to optimize the use of cloud resources in order to obtain financial success in their cloud projects. When the PaaS users contract the use of shared resources (CPU, memory, databases, or web server, among others), this problem is more apparent.We can cite the example of a PaaS provider that shares resources from a third-party IaaS provider. The PaaS provider need to allocate as many users as possible in the same resource without losing quality of service (QoS), ensuring acceptable response time and usability in accordance with defined service level agreements (SLAs). The optimal allocation problem is similar to the traditional "knapsack problem", known to be NP-complete, and whose resolution requires specific heuristics to enable the computational implementation in acceptable time. We need to identify the heuristics that can provide the best results for this problem.The complexity is even greater when we need to optimize total cost of infrastructure use. When IaaS plans hired by the PaaS provider uses automatic elasticity features (auto scaling), the potential costs are virtually unlimited, and any savings in each operation may represent a significant value.

In this paper, we focus on modeling and reducing cost of elasticity of cloud services. Elasticity, also known as dynamic provisioning, “has become one of the most important features of a cloud computing platform” [3]. By using this feature, application owners can scale up and down the resources used basedon the computational demands of their applications, and pay only for the resources they actually use. Elasticity places new challenges in resource management, as pointed in [4], and makes it harder to estimate costs, thus adding more complexity to the decision making process.

Our main work contribution resides in the framework proposal and findings from simulation scenarios while investigating savings using spot prices. The PN model to estimate cost savings uses a particular purchasing option for virtual machines named spot instances [5]. This purchasing option is currently supported by Amazon Web Services (AWS), the leader in public IaaS market, according to Gartner´s analysts [6]. Spot instances work exactly the same way as any other running EC2 virtual machine. The difference lies in the pricing scheme: the hourly price is not fixed; in fact clients bid on how much they are willing to pay for them. AWS dynamically defines Spot Price, which varies in real-time based on supply and demand. If the client bid is above the current Spot Price, then the instance is started. If Spot Price changes, and rises above the client bid, then the instance is terminated by AWS. In this paper, we will refer to instance as any type of virtual machine that can be rented in a public cloud.

II. Literature review

A Colored Petri Net [7] (CP-net or CPN) is a graphical language for constructing models. A CPN is a discrete-event modeling language combining the capabilities of Petri nets with the capabilities of a high-level programming language [8]. We used CPN Tools [9] to design the hierarchical Petri nets to compose our framework. CPN Tools also supports the inclusion of timing information to the framework.

We conducted a literature review, where we looked for ‘Cloud Cost Model’. We selected 43 papers that seemed more relevant to our study. An interesting study related to comparison of on-premises and cloud services is found in [10]. The authors compare costs and overhead for HTC jobs in two environments: a public cloud and a desktop cluster of non-dedicated resources. Their cloud cost model considers hours of use of instances and upload/download data. They point that start of a billing period varies between providers.

Some, including AWS, charge from the start of the wall-clock hour in which the instance was started – billing from 7 pm for an instance started at 7:59 pm whilst others charge from the time the instance was actually started. Their on-premise cost model considers factors like cost of hardware acquisition, cost of providing technical support for the desktop cluster, charges incurred for carbonemission, and energy cost (per kWh). They propose six different policies for cost savings in the cloud: P1 - limiting the maximum number of Cloud instances, P2 - merging of different users' jobs, P3 - instance keep-alive, P4 - delaying the start of instances, P5 - removing the delay on starting an instance, and P6 - waiting for the start of the next hour.

The work in [11] compared costs of HPC (high performance computing) on-premises and in the cloud. The simplified total cost of the on-premises cluster mainly depends on the purchase of the hardware, the maintenance and operation of the cluster, and its energy consumption (which can be lowered by turning off idle nodes). For the cloud cost model, they focus on hours of instances use and analyze factors like purchasing options.

Elasticity in multi-tier cloud application is analyzed in [3]. The authors proposed an algorithm that relies on online monitors to detect the changes in workloads and perform corresponding scaling in each tier. The algorithm was designed to measure the cost spent in adding a server divided by the decreased response time because of this addition. Hence this criterion was called the consumed cost/decreased response time (CC/DRT) ratio.

Some studies refer to cost optimization using linear programming techniques [12], using cache as a service (CaaS) to reduce I/O costs and improve performance [13]. The study in [14] explored factors that affected chargeback for cloud services, mainly acceptability and effectiveness, and presented interesting insights on qualitative issues in cloud services use.

Spot instances were studied in[15] to characterize their behavior through statistical models. The authors present probability density functions (pdf) for Spot Price and interval for price spot change. Another study of Spot Prices [16] proposed a framework for bidding on spot prices in order to achieve monetary advantages and still comply with SLA regulations.

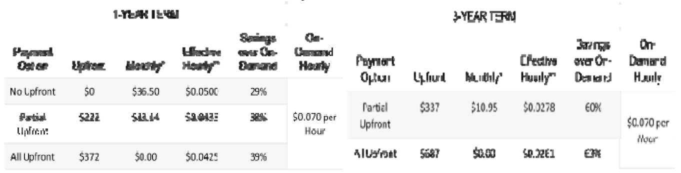
Petri nets were used in [17] as a tool for stochastic generation of dependability and cost models for representing cloud infrastructures.

To the best of our knowledge, there is no study to provide a model to estimate cost savings using spot instances.

III. Our framework

Cloud providers usually charge customers in a pay-per-use basis, that is, the customer pays for each hour (or minute, or month) that the machine stays turned on. Each IaaS provider has its own billing model for virtual machines use. In this paper, we investigated Amazon Web Services (AWS) current purchasing options. AWS is the leader in IaaS market, and currently offers three purchasing options:

- **On demand**: charges are for each hour the virtual machine (named instance) is turned on. There is no upfront investment, and no commitment of use. It is the simpler way of use and pay, but usually the most expensive hourly rate;
- **Reserved**: customers pay for the period of reservation instead of hours of use. Payment options may be: no upfront, partial upfront or all upfront. Figure 1 illustrates prices for US-east region and m3.medium instance size. AWS presents an hourly estimate of the cost to compare prices to the On demand option; however, reserved instances are paid by the period (month, year) and not by hour of use. This means that if you purchase a reserved instance for a month, it makes no difference if it stays on or off, the price will be the same. Reserved instances prices can be equivalent to on premises cost of operating a local server [18];
- **Spot**: charges are for hours of use, similar to *on demand* option, however, the hourly price is not fixed. Clients bid on how much they are willing to pay for the hour. AWS dynamically defines Spot Price, which varies in real-time based on supply and demand. If the client bid is above the current Spot Price, that the instance is started. If Spot Price changes, and rises above the client bid, then the instance is terminated by AWS.



1-YEAR TERM

Payment Option	Upfront	Monthly*	Effective Hourly**	Savings over On-Demand	On-Demand Hourly
No Upfront	$0	$36.50	$0.0500	29%	$0.070 per Hour
Partial Upfront	$222	$13.14	$0.0433	38%	
All Upfront	$372	$0.00	$0.0425	39%	

3-YEAR TERM

Payment Option	Upfront	Monthly*	Effective Hourly**	Savings over On-Demand	On-Demand Hourly
Partial Upfront	$337	$10.95	$0.0278	60%	$0.070 per Hour
All Upfront	$687	$0.00	$0.0261	63%	

Figure 1. Reserved Instances Prices for US-east Region and m3.medium instance size.

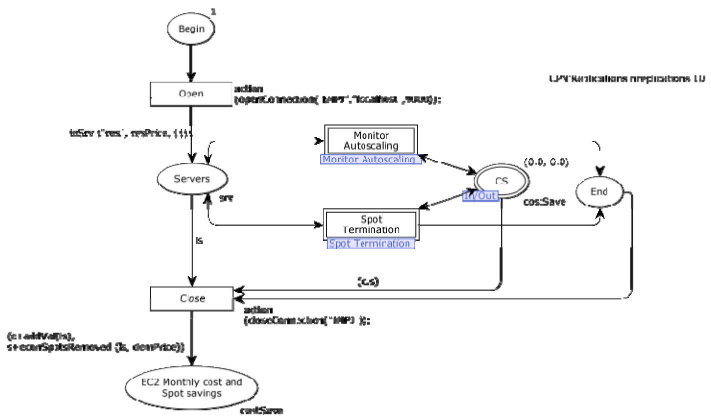


Figure 2. CPN Model for Instance use Simulation

To investigate how the use of spot instances can help in cost reduction, we created a CPN model hierarchically organized in modules that will compute:1) the monthly cost of all running instances and 2) savings by using Spot instances. In our model, there will be one (could be more, if necessary) reserved instance, that will remain always on, to guarantee availability of the service all the time. Since it will always be on, the reserved option is the most cost effective option. The other instances will be turned on and off whenever needed, by monitoring the demand for servers, simulating the auto scaling process. This way, the model will simulate elasticity of server use. Figure 2 illustrates the main CPN model, that will be decomposed into four subnets: Monitor AutoScaling, Scale Up, Scale Down and Spot Termination. We will discuss these nets in detail.

CPN Model: Instance Use Simulation

This model represents the proposed mechanism for using Spot Instances for cost savings in elasticity. There are 4 input parameters to the model:

- Hourly price for On Demand instance, represented by the constant *demPrice* in the model.
- Hourly price for Reserved instance, represented by the constant *resPrice* in the model.
- Hourly price for Spot instance. These values will be obtained dynamically by using AWS API, during the simulation period. To accomplish that, we use a special programming interface for Java.
- demand() function: This function represent the demand for servers in the auto scaling process. It will return the number of servers needed at some point in time. It must be customized when using the model. Figure 3 presents an example of demand() function use. In this case, the function will return 1 server needed for nighttime (23h to 6h) and will return 1 or more servers in daytime (6h-22h). To compute how many extra servers will be needed, we use a Normal distribution with average 3 and standard deviation 0.5. This function should be adapted to reflect each business scenario of needs for extra servers in auto scaling.

```
fun demanda() =
 let
   val h = intTime() mod 24
   val extra = normal (3.0, 0.5)
 in
   if h<7 orelse h> 22 then
     1
   else
     1+floor(extra)
 end;
```

Figure 3. Customizable demand() function example

There are 2 output values of the model:

- Total monthly cost of EC2 instances use, including charges for all instances (Reserved, Spot and On Demand)
- Total savings obtained by using Spot instances compared to On Demand instances.

The CPN model will use the following elements:

- Color *server*: represent one instance currently turned on. It is a tuple of 3 information (type, price, time), where type can be *res*, *dem* or *spot*, to identify the purchasing option (reserved, on demand or spot), price is the hourly price charged for that instance, and time is the start time of use of the instance.
- Color *srv*: list of active instances, representing all servers that are currently turned on.
- Color *costSave*: a tuple of 2 real numbers (r1, r2) where r1 represents the total monthly cost and r2 represents the total savings.
- Place *Begin*: contains one timed token to start the simulation process at model time 0.
- Transition *Open*: will open the connection with the Java programming interface and initialize the list of servers, including a reserved instance in the list.
- Function *inSrv (type, price, list)*: will insert one server of given *type* and *price* to the *list* of active servers.
- Transition *Monitor Autoscaling*: a substitution transition to model the auto scaling process (turning servers on and off as needed), it will be discussed in detail when we present the corresponding subnet.
- Transition *Spot Termination*: a substitution transition to model the spot termination process (turning servers off due to changes in spot prices), it will be discussed in detail when we present the corresponding subnet.
- Place *cs*: used as a temporary space to add the cost of using one server when it is turned off. At this moment, one can compute the total hours of use of this server and multiply by the hourly price. It will be discussed in detail when we present the *Monitor Autoscaling* and *Spot Termination* subnets.
- Place *end*: will receive a token when simulation reaches a predefined time (720 hours = 24 hours * 30 days), representing the end of the month being analyzed.
- Transition *Close*: will close the connection with the Java programming interface and finalize the simulation process, modeling the action of turning off all servers.
- Function *addVal(list)*: will compute the cost of use of each server in the list, multiplying hours of use by hourly price. Then, it will add cost for all servers.
- Function *econSpotsRemoved (list)*: will compute the savings of using each spot instance in the list, multiplying hours of use by (on demand hourly price - spot hourly price). Then, it will add savings for all instances.
- Place *EC2 Monthly cost and Spot savings*: at the end of simulation, its marking (c,s) will represent the total cost of server use (c) and savings using spot instances (s), the output values of the model.

Subnet: Monitor AutoScaling

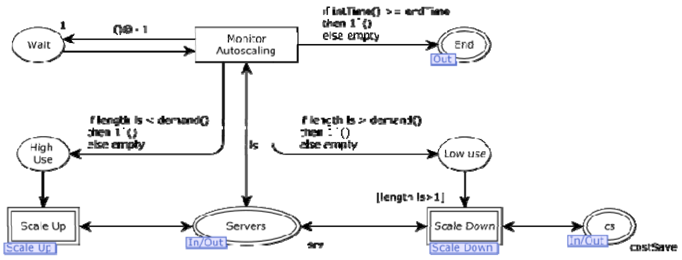


Figure 4. Monitor Auto Scaling subnet

This subnet models the auto scaling process. Figure 4 illustrates the CPN model, that implements the transition Monitor Auto Scaling in main net. It will be executed every hour, and monitor if there is need for turning on or off servers. It will compare the need of servers at this time (given by demand() function) with the number of active servers (length of list of active servers). When demand is greater, it will put a token in place *High Use*, which will drive *Scale Up*process, that will be discussed in its own subnet. When demand is smaller, it will put a token in place *Low Use*, which will drive

Scale Down process, that will be discussed in its own subnet as well. If demand is neither greater nor smaller, it will only wait until next hour, by putting a token in *Wait* place that will only be available in the next hour. It also checks the end of simulation, and in this case, puts a token in place *End*.

Subnet: Scale Up

This subnet will model actions needed to add servers to our server list with the least cost possible. Figure 5 illustrates subnet Scale Up.

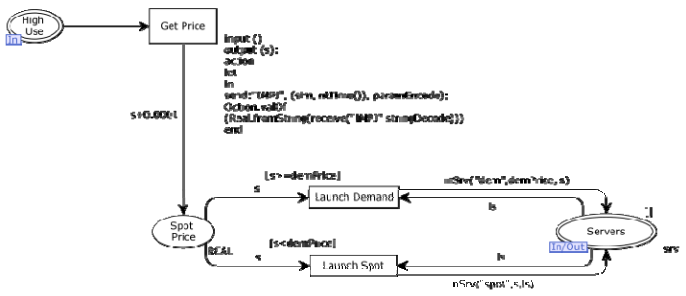


Figure 5. Scale Up subnet

Table 1. Parameters for obtaining spot prices in AWS

Simulation	Region	AZ	OS	Instancetype
1	South America	sa-east-1a	Windows	m3.medium
2	South America	sa-east-1a	Windows	m3.2xlarge
3	South America	sa-east-1a	Linux/UNIX	m3.medium
4	South America	sa-east-1a	Linux/UNIX	m3.2xlarge
5	US-East	us-east-1c	Windows	m3.medium
6	US-East	us-east-1c	Windows	m3.2xlarge
7	US-East	us-east-1c	Linux/UNIX	m3.medium
8	US-East	us-east-1c	Linux/UNIX	m3.2xlarge

The policy implemented by this subnet is to obtain the current Spot price, place a bid a little above, and verify which has the lowest price, the spot instance or the on demand instance. It is important to notice that on-demand instances are always available, this way this subnet will always add one server to our server list. The CPN model will use the following elements:

- Place *High use*: has a token when scale up is needed
- Transition *GetPrice:* will compute an optimal bid for a spot instance. To do that, it will use the Java interface to obtain the current spot price for the model time passing parameters *(simulation#, modelTime)* where simulation# is a simulation number, that represent a set of parameters needed to get spot prices, and model Time represents the day of month and time to obtain spot price. Tables 1 and 2 illustrate input parameters of Java interface.
- Place *Spot Price*: holds the bid for the spot instance, which will be the spot price (a real number) returned by the Java interface plus $0.0001. This way, we ensure that our bid is high enough to obtain a spot instance, at the least possible cost.
- Transition *Launch Demand*: fired when marking in *Spot Price* place is higher than on demand price. In this case, it is not interesting to use spot instance, since it is currently more expensive. It will insert a server purchased using "on demand" option in the list of active servers, with the corresponding *demPrice* (input parameter). To do that, it will use *inSrv* function (type, price, list), that will also save the model time when the server was inserted in the list and update the list of active servers.
- Transition *Launch Spot*: fired when marking in *Spot Price* place is lower than on demand price. In this case, it is interesting to use spot instance, since it is currently cheaper. It will insert a server purchased using "spot" option in the list of active servers, with the corresponding price, that is, the bid. It will use *inSrv* function.

Table 2. Model time and corresponding time of day in simulation

Model Time	Day of Month	Hour of day
1	1	0
2	1	1
…	…	…
24	1	23
25	2	0
…	…	…
720	30	23

Subnet: Scale Down

This subnet will model actions needed to remove servers from our server list, selecting first the servers with the greatest cost possible, to keep using the cheaper ones. Figure 6 illustrate subnet Scale Down.

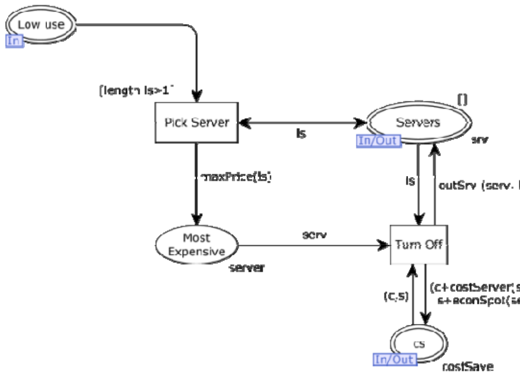


Figure 6. Scale Down subnet

The policy implemented by this subnet is to select the server with the maximum price charged hourly, and turn it off. It also implements the policy of always leaving ON the server covered by a reservation (where the client paid an upfront fee to get better prices for the hourly charges). This way, even when there is low utilization, there will be one server available. This was built to model real world scenarios in business applications, where the application is available 24 hours a day, always. It can also model the situation where the 'reserved' server is actually an internally hosted server, as in hybrid clouds. In fact, it is noticeable that in terms of cost, in some scenarios the cost of a 'reserved' server is equivalent to an internally hosted server [18].

The CPN model will use the following elements:

- Place *Low use*: has a token when scale down is needed
- Transition *Pick server*: will pick the most expensive server, to be turned off. It will use function maxPrice(list) that will select the server with the lowest hourly price. This function excludes reserved instances, because turning them off will make no difference in the final cost, since they are always charged for the whole period, regardless of being used or not.
- Transition *Turn Off*: will exclude from the server list the selected server. Will also compute total cost for using that

instance and the savings by using spot instances compared to on-demand instances, using functions CostServer and EconSpot

- Function *CostServer* will compute cost of using the server using the expression: Cost of use = (hours of use) * (hourly price)
- Function *econSpot* will compute savings using the expression: Savings = (hours of use) * ((hourly price for on-demand) – (hourly price for spot)). It is important to notice that savings will only be computed when turning off spot instances, since they will be zeroed when turning off on-demand instances (hourly price for on-demand = hourly price)

Subnet: Spot Termination

This subnet will model regular verification of Spot price market. As mentioned before, spot instances are automatically terminated by AWS when Spot price rises above the price the client is currently paying for them. Figure 7 illustrates subnet Spot Termination.

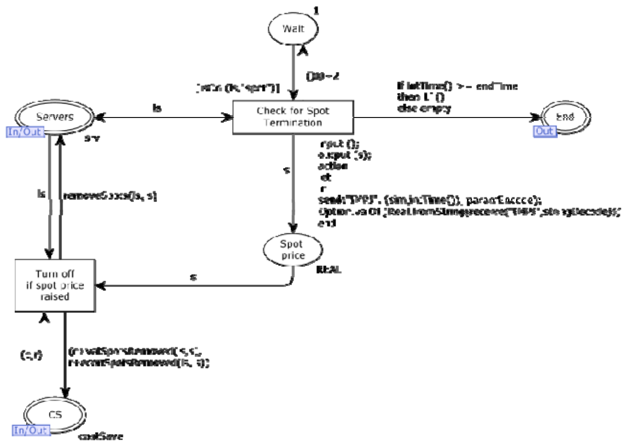


Figure 7. Spot Termination subnet

The CPN model will use the following elements:

- Transition *Check for Spot Termination (simulation#, modelTime)*: will be executed every two hours, to monitor spot prices and simulate termination of spot instances. To do that, it will use the same Java interface to obtain the current spot price that was used in transition GetPrice in subnet Scale Up. It will only be fired when there is at least one spot instance in the list of active servers. After fired, It will wait for 2 hours, by putting a token in *Wait* place that will only be available after 2 hours. It also checks the end of simulation, and in this case, puts a token in place *End*.
- Place *Spot Price*: holds the current spot price returned by the Java interface.
- Transition *Turn Off if spot price raised*: will check the list of active servers looking for spot instances with hourly prices lower than the current spot prices. These instances will be terminated (turned off). It will use valSpotsRemoved() and econSpotsRemoved() to compute the cost of use of the terminated instances and savings as computed in *Scale Down* subnet. The only difference is that *Turn Off if spot price raised* may remove several servers from the list of active servers at once. In our model, the termination of these instances may cause several *Scale Up* transitions to be fired, if utilization levels are still high.

IV. Case Study and Result analysis

We did a case study to validate our framework proposal. The experimental design of the simulations followed a 2^k factorial design. Since we used AWS Spot Instances, we examined the factors that affected EC2 prices:

- Region-AZ: AWS currently offers their services in 9 different geographic locations around the world, named Regions: US-East (N. Virginia), US-West (Oregon), US-West (N. California), EU (Ireland), EU (Frankfurt), Asia Pacific (Singapore), Asia Pacific (Tokyo), Asia Pacific (Sydney), and South America (São Paulo). Each region has at least 2 availability zones (AZs), which are datacenters in different locations connected through low-latency links. Each region has its own pricing table for On-demand and Reserved Instances. Spot Instances price may also vary by AZs in each region.
- Operating System: AWS has different prices for hours of use of EC2 instances depending on the O.S: Linux/Unix, SUSE Linux and Windows.
- Instance type: EC2 prices vary upon the size of the instance. For our experiments, we considered m3.medium (1 vCPU, with computational power of 3 ECU, 3.75 GB of memory and 1HD type SSD with a storage capacity of 4GB) and m3.2xlarge (8vCPU, with computational power of 26 ECU, 30 GB of memory and 2HD type SSD with a storage capacity of 80 GB)

Table 4 illustrate factors and levels in the experimental design of our simulations

Table 4. Factors and levels

Factor	Levels	Selected for Experiment
Region-AZ	Over 20	South America (1a), US-East (1c)
OS	3	Windows, Linux/Unix
Instance type	Over 20	m3.medium, m3.2xlarge

In each of the selected $2^3 = 8$ scenarios we ran 10 simulations to obtain statistical information. The evaluated scenarios are shown in Table 5.

Table 5. Simulations using framework

					Hourly price		
Scenario	Region	AZ	OS	Instance type	On demand	Reserved (1 year all upfront)	Upfront investment
1	South America	1a	Windows	m3.medium	0,1580	0,1410	$1.235,00
2	South America	1a	Windows	m3.2xlarge	1,2650	1,1205	$9.816,00
3	South America	1a	Linux/UNIX	m3.medium	0,0950	0,0509	$446,00
4	South America	1a	Linux/UNIX	m3.2xlarge	0,7610	0,4063	$3.559,00
5	US-East	1c	Windows	m3.medium	0,1330	0,0855	$749,00
6	US-East	1c	Windows	m3.2xlarge	1,0640	0,6809	$5.965,00
7	US-East	1c	Linux/UNIX	m3.medium	0,0700	0,0425	$372,00
8	US-East	1c	Linux/UNIX	m3.2xlarge	0,5600	0,3412	$2.989,00

Savings using spot instances

Figure 8 presents simulation results for savings. For the South America region, the simulation returned higher savings for m3.medium instances. In fact, for m3.2xlarge instances, there was no savings at all when using Linux/UNIX. That happened because in the whole period of simulation spot prices were higher than on demand prices ($1.2240 for spot and $0.7610 for on demand). In this case, our model did not use spot instances and consequently there were no savings.

For US-East region, simulation returned higher savings (up to 67%) than in South America region, and the highest value was found in m3.2xlarge instances. This happened because spot prices remained consistently low during the simulation period ($0.0641 for spot and $0.5600 for on demand).

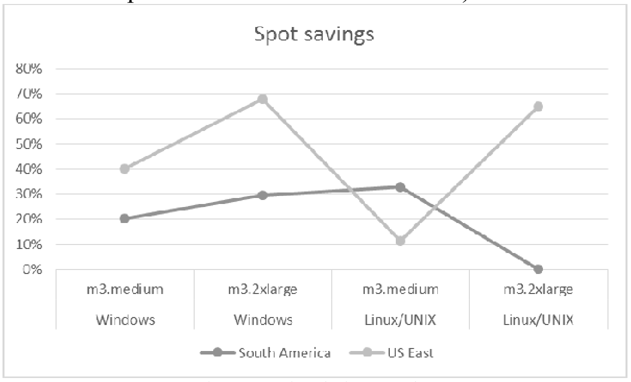


Figure 8. Simulation results

Face validity

Validating a framework, such as we have presented here, is a multi-year effort. To start however, we applied a face validity questionnaire to 20 IT managers. The questionnaire included three questions, each of which leads to a hypothesis to be tested. We presented the framework theory to managers before the evaluation. Statistical inference was used to test the hypotheses (binomial statistical test with a 5% significance level). Our initial results are shown in Table 8. Face validity appears to be established in the three dimensions. Managers considered the framework to be "useful", "effective" and "preferable" to their current way of decision making. We plan to expand and repeat this test in a future work.

Table 8. Hypotheses to test theory face validity

Hypothesis	% who agree	Is there statistically significant evidence to accept hypothesis?
Preference: Manager prefers the framework to the current process	100	yes
Utility: Manager considers the framework useful	100	yes
Effectiveness: In modeling a business scenario, manager can identify value elements in cost reducing to decision-making support	90	yes

V. CONCLUSION AND FUTURE WORK

Cloud Managers need solutions to support an effective cost reduction and resource optimization, to support the decision-making process.

In this work we proposed a Petri net framework to evaluate cost reduction using spot instances. We proposed a set of policies in auto scaling that can help in cost reduction of cloud services. Our main contributions were the framework proposal and the simulation scenarios evaluation.

Our preliminary studies indicated that using spot instances in auto scaling process may help reduce cloud services costs. We proceeded an initial face validity exercise, where results were promising. Managers evaluated our framework as useful, preferable and effective. The results of our presented framework can be used by managers as a criterion in decision-making about cloud computing adoption. As a threat to validity, we can cite the limited number of considered scenarios and simulations. Albeit these limitations, our initial results were promising.

This paper focus was the cost reduction treatment. We plan to expand and use our framework to support decision-making in IaaS, PaaS and SaaS scenarios evaluation. As future work, we plan to execute extensive simulations to complete our framework validation, and a possible extension of the framework to handle SLA violations and fines.

REFERENCES

[1] NIST - National Institute of Standards and Technology. NIST Definition of cloud computing. Gaithersburg, MD, 2009.

[2] Ribas M., Lima A. S., De Souza J. N., Moura A., Sousa F. R. C., Fenner G. Assessing Cloud Computing SaaS adoption for Enterprise Applicationsusing a Petri net MCDM framework, Ninth Business-driven IT Management Workshop-BDIM, pp. 1-6, 2014.

[3] Han, R. et al. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. Future Generation Computer Systems, 2014.

[4] Manvi, S. S., Shyam, G. K. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. Journal of Network and Computer Applications, pp. 424-440, 2014.

[5] Amazon. Amazon EC2 Spot Instances. Fonte: Amazon Web Services: http://aws.amazon.com/ec2/purchasing-options/spot-instances. Accessed in December, 2014.

[6] Gartner Group. Magic Quadrant for Cloud Infrastructure as a Service. Fonte: http://www.gartner.com/technology/reprints.do?id=1-1UKQQA6&ct=140528&st=sb. Accessed in December, 2014.

[7] Peterson, J.L. Petri net theory and the modelling of systems. Prentice Hall, 1981.

[8] Jensen, K., Kristensen, L. Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer, 2009.

[9] CPN Group. (2013). CPN Tools. Fonte: http://cpntools.org/.Accessed in November 2013.

[10] McGougha, A.S. et al. Comparison of a cost-effective virtual Cloud cluster with an existing campus cluster. Future Generation Computer Systems, 2014.

[11] Alfonso, C, et al. An economic and energy-aware analysis of the viability of outsourcing cluster computing to a cloud. Future Generation Computer Systems, 2013.

[12] Malawski, M. et al. Cost minimization for computational applications on hybrid cloud infrastructures. Future Generation Computer Systems, 2013.

[13] Han, H. et al. Cashing in on the Cache in the Cloud. IEEE Transactions on Parallel and Distributed Systems, 2012.

[14] Baars, T. et al. Chargeback for cloud services. Future Generation Computer Systems, p. http://dx.doi.org/10.1016/j.future.2014.08.002, 2014.

[15] Javadi, B. et al. Characterizing spot price dynamics in public cloud environments. Future Generation Computer Systems, 2013.

[16] Tang, S. et al. A Framework for Amazon EC2 Bidding Strategy under SLA Constraints. IEEE Transactions on Parallel and Distributed Systems, 2014.

[17] Sousa, E. et al. A Modeling Approach for Cloud Infrastructure Planning Considering Dependability and Cost Requirements. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2014.

[18] e-fiscal. Computing e-Infrastructure cost estimation and analysis - Pricing and Business Models. In: Financial Study for Sustainable Computing e-Infrastructures, 2013.