Dependable Horizontal Scaling Based On Probabilistic Model Checking

Athanasios Naskos*, Emmanouela Stachtiari*, Anastasios Gounaris*, Panagiotis Katsaros*,
Dimitrios Tsoumakos‡, Ioannis Konstantinou† and Spyros Sioutas‡
*Aristotle University of Thessaloniki, Greece, Email: {anaskos,emmastac,gounaria,katsaros}@csd.auth.gr
†National Technical University of Athens, Greece, Email: ikons@cslab.ece.ntua.gr
‡Ionian University, Greece, Email: {dtsouma,sioutas}@ionio.gr

***Abstract*—The focus of this work is the on-demand resource provisioning in cloud computing, which is commonly referred to as cloud elasticity. Although a lot of effort has been invested in developing systems and mechanisms that enable elasticity, the elasticity decision policies tend to be designed without quantifying or guaranteeing the quality of their operation. We present an approach towards the development of more formalized and dependable elasticity policies. We make two distinct contributions. First, we propose an extensible approach to enforcing elasticity through the dynamic instantiation and online quantitative verification of Markov Decision Processes (MDP) using probabilistic model checking. Second, various concrete elasticity models and elasticity policies are studied. We evaluate the decision policies using traces from a real NoSQL database cluster under constantly evolving external load. We reason about the behaviour of different modeling and elasticity policy options and we show that our proposal can improve upon the state-of-the-art in significantly decreasing under-provisioning while avoiding over-provisioning.**

I. Introduction

Clouds are able to adapt to the actual user requirements utilizing on-demand resource provisioning, which is commonly referred to as elasticity. We adopt the standard elasticity definition proposed in [1]: "*Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.*" Elasticity may be manifested in different forms and can refer to the size, the location or the number of virtual machines (VMs) employed. Examples of these three elasticity types are the allocation of more memory to a VM (vertical scaling), moving a VM to a less loaded physical machine (migration) and increasing the number of VMs (horizontal scaling) of an application cluster, respectively. We exclusively focus on automated elasticity approaches, and we especially target elasticity in the form of horizontally scaling the number of application VMs. Increasing or decreasing the number of VMs is a key element in adapting to dynamically changing volumes of user requests, e.g., as typically occurs in cloud databases, which is the scenario we use in our evaluation.

There have been numerous proposals for elasticity, which differ in several dimensions including the form of the elasticity they support, the underlying objectives driving the elasticity actions and the decision making policy (e.g., reactive or proactive), such as [2], [3], [4], [5], [6]. However, elasticity proposals tend to work on a best-effort basis without being able to guarantee their adequacy under the expected workload scenarios. The main aim of our proposal is to make a decisive step towards more formalized and dependable elasticity decision policies. Dependability refers to the fact that elasticity actions should be selected according to the results of continuous verification of elasticity aspects that are of user interest, including the resulting system utility and the probability to experience SLA violations. At a higher level, we view the elasticity problem as a specific instance of autonomic computing [7], for which the need for coupling continuous verification when responding to environmental changes has already been identified [8]. To this end, we adopt a formal verification approach, as a means to apply mathematical reasoning for providing correctness guarantees for the elasticity policy and we employ a mature model-based verification technique, namely probabilistic model checking [9].

In brief, our approach is twofold. First, we present expressive models of elasticity actions and second, we leverage them for devising concrete policies that can take elasticity decisions. The mathematical modeling framework we build upon is Markov Decision Processes (MDPs), because MDPs can capture both the non-deterministic and probabilistic aspects of the problem. Given the current state, non-determinism is due to the applicability of several possible elasticity actions, which may lead to different future system states. In addition, the probabilistic behaviour allows us to take into account the effects of the unpredictable environment's evolution, which may result in reaching different states even for the same action applied under the same current conditions.

We use the PRISM probabilistic model checker tool [10], because it supports both the specification of MDP system models at a high-level and the easy specification of probabilistic reachability and reward-based properties via the PCTL logic language [9], which are amenable to model checking. We introduce properties that, on the model level, yield optimal decisions for system reconfigurations aiming to maximize the system utility, under user-specified

probabilistic guarantees. We show how our decision making policy can be incorporated into existing systems. Finally, we compare our policy proposals against the policies of the Amazon's EC2 manager and the novel Tiramola system[1], which supports elastic scaling of NoSQL databases [2]. In summary, the main contributions are:[2]

1) We present a concrete approach to employing continuous online quantitative verification for taking runtime elasticity decisions, with a view to making them more dependable. Our approach is well-founded and is based on extensible, automatically generated and dynamically instantiated MDP models.
2) We present modeling variations and related elasticity decision policies that aim to maximize user-defined utility functions; moreover, our decisions are subject to quantitative analysis.
3) We conduct elasticity experiments using real log traces from an elastic NoSQL cluster under constantly evolving external load, which is a particularly demanding elasticity evaluation scenario [12]. Based on the results, we show that we can improve on under-provisioning using probabilistic model checking-based elasticity policies while avoiding over-provisioning.

The remainder of this paper is structured as follows. In Section II, we present our approach to elasticity decision making. We introduce the underlying MDP models and the elasticity policies that are built on top of their runtime instantiations. We also explain how the PRISM tool can be used for this purpose. Next, we discuss how our approach can be incorporated into existing systems. We evaluate our decision making solutions in Section IV, we refer to the related work in Section V and we present future extensions of our work and conclusions in Section VI.

II. Elasticity based on probabilistic model checking

Probabilistic model checking is a formal verification technique for the modeling and analysis of stochastic systems [10]. In our work, probabilistic models are used in the decision making process, to specify, drive and analyse cloud resource elasticity. By utilizing probabilistic models, we are able to capture the uncertainty in systems' elasticity. In order to additionally infuse non-determinism for representing the multiple possible elasticity actions, we resort to MDP models, which form the basis of our approach. On top of our MDP models, we build policies for elasticity decisions with the help of the PRISM probabilistic model checker [10]. While our main objective is to render elasticity decision policies more dependable, through the avoidance of under-provisioning, our principled approach is capable of yielding higher utility than simple reactive techniques, where higher utility is linked with lower over-provisioning in our work.

Next we discuss how we model the elasticity actions and present exact approaches to taking elasticity decisions. In general, we periodically monitor the incoming load and the system state; also, we periodically activate the decision policy, and we call such an activation an elasticity step. The monitoring frequency is either equal to or higher than the decision making frequency. An elasticity step is further split in the following three sub-phases:

1) Dynamically instantiate a model according to the current incoming load and the log measurements.
2) Verify the model online to reach elasticity decisions.
3) Take elasticity actions. Suspend the run of the next elasticity step until the system stabilizes.

The first subphase is the most important one. The model is dynamically instantiated so that, in each step, it can describe the expected behaviour according to the current environmental conditions. In our implementation, we assume that those conditions are defined by the (external) incoming load λ of requests. We assume that the system that sets the elasticity decision policy keeps log measurements in order to be able to evaluate the utility functions given the current value of λ as explained below. Also, the elasticity loop is suspended for some period after each action to allow the system to stabilize. In applications with stateless VMs, this period is very short; however, for applications running VMs with partitioned databases, this period can be 5-10 minutes (or even higher), since each new VM needs to receive data from running VMs in order to become operational. Finally, elasticity decisions are typically bounded according to user-specified limits, so that not too many VMs are added or removed in a single step. When the upper bound takes into account the maximum load change in a step (as we do in this work), there is no compromise regarding elasticity efficiency [13], while it offers protection from over-reacting; the need for a lower bound may also stem from the replication factor in NoSQL databases, which implicitly specifies how many nodes can be removed in a single step without possible data loss.

A. Elasticity Models

MDPs provide a mathematical framework for modeling decision making in situations, where outcomes are partly random and partly under the control of a decision maker [14]. This condition fits well into our problem domain, where we need a) to take decisions by choosing among multiple options, i.e, adding or removing or maintaining the number of active VMs and b) to maximize a utility function that quantifies the value of each system state, which is constantly evolving and hard, if not impossible, to be accurately predicted. MDP model verification is used in this work to guide the elasticity. In the remainder of this section we present the main rationale of our modeling approach.

[1]Best-paper award in 2013 IEEE/ACM International Conference on Cluster, Cloud and Grid Computing.

[2]A preliminary extended version with fewer modeling and decision options but complementary experiments is in [11].

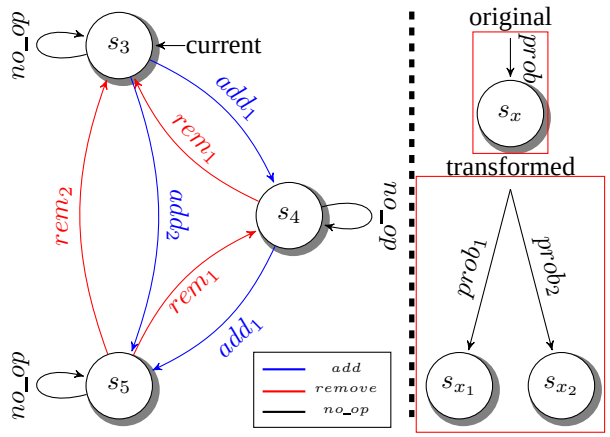


Figure 1: MDP model overview.

MDPs are specified by defining their states, actions, transition probabilities and rewards. In our model, each state corresponds to a different cluster size, where the size equals to the number of active VMs, vms_num. For readability reasons, we denote a state as $s_{[vms\_num]}$. There are three types of possible actions on every state: 1) add for VM additions, 2) rem for removals, and 3) no_op for no operation. For every distinct number of VM additions or removals (ex. add_1, rem_2) there is a separate action, and the corresponding transitions between two states through the same action have aggregate probability 1.

The MDP associates a reward value to each state and action. State and action rewards are calculated based on user-specified utility functions. When the model is verified at runtime, the utility at state $s_{[vms\_num]}$ essentially describes the expected behaviour of the system when there are vms_num active VMs.

Figure 1(left) illustrates a simplified instance of the MDP model, where the states represent the number of active VMs. The edges represent the possible actions: 1) $add_{new\_vms\_num}$ (blue arrow), 2) $rem_{removed\_vms\_num}$ (red arrow), and 3) no_op (black arrow). In this example, the maximum number of VMs allowed to be added or removed in every step is 2, while the current number of active VMs is 3 (s_3 state). The action type is labelled on top of every transition ($[add_x/rem_x/no\_op]$).

The model of Figure 1 assumes that each state is associated with a single reward value, but it can easily become more elaborated to reflect additional aspects. Let us now assume that the reward function depends on system latency and throughput. However, for the same number of VMs, the latency and throughput may vary significantly, due to factors that are external to our model. This leads to an undesirable situation, where the state reward does not reflect the actual system behaviour well, and there is a high probability that the system may end up in a state that significantly diverges from the expected one. To ameliorate this, we can increase the number of model states that correspond to a specific number of VMs, so that, each state corresponds to a distinct representative expected behaviour for the specific number of active VMs. Moreover, we extend the model so that it explicitly covers the probabilities of encountering each of the new states. As it is presented in Figure 1(right), there are two model states (s_{x_1}, s_{x_2}) for a single size x, and the transitions are enriched with the probabilities of encountering each of those two states through the same action; $prob = prob_1 + prob_2$ in the figure. Thus, when a MDP solver examines possible actions to maximize the total reward, it can better capture the fact that the behaviour of the system is non-deterministic and unpredictable. Nevertheless, the higher expressivity of the model comes at the expense of larger state space size, compared to the simpler model. However, according to Section IV-D, the additional overhead is negligible. More details about the number of states per VM number are given in Section II-B2, where the notion of clusters of log measurements is added to our model.

B. Policies for Elasticity Decisions

There are several options to analyze the MDPs. We distinguish between indirect (based on reinforcement learning) and direct methods (based on dynamic programming).

Indirect methods are exemplified by the Tiramola approach, which relies on online training and convergence of action-value functions, which, in turn allows to attain optimal policies through greedy actions and a Q-learning-based reinforcement learning approach. Exact details are provided in [2].

The direct methods analyze MDPs per se. In our approach we use the PRISM tool to this end, because, as will explained later, we do not only solve it online but we also perform online property verification.

1) On solving MDPs directly: A challenge in the direct MDP solutions is to define conditions to terminate the verification process in order to allow for meaningful quantitative verification (i.e., ensuring finite rewards). We propose two distinct ways to perform this task:

- Bounded By Action (*BBA*): In our model, the verification can be terminated on every state if that state is reached through a no_op action, because the latter denotes that no change in VMs is beneficial. In addition, once the first action is an add (resp. rem) one, we allow only VM additions (resp. removals), i.e., we do not allow mixed add and remove state transitions, which are harmful in practice. Consequently, every accessible state is visited and its reward is computed at most once.
- Bounded By Steps (*BBS*): According to this option, when a pre-specified number of transitions (steps) is reached, the verification terminates. Every possible action is allowed in every step, including multiple actions of the same type (e.g., $no\_op \rightarrow no\_op$). This is the most broadly used way to terminate verification in traditional model checking.

The solution of an MDP consists of finding a sequence of transitions that lead to reward maximization. In our approach, we assign rewards only to states; assigning rewards to actions is left for future work. Based on the selected verification termination condition presented above, there are two ways of reward manipulation:

- *Instantaneous rewards:* if the *BBA* way is used, then instantaneous rewards for every distinct reachable state are computed; those instantaneous rewards imply that the reward of only the final state reached is of significance [9].
- *Cumulative rewards:* if the *BBS* way is employed, then a cumulative state reward is used to derive the total reward of an examined path. On every step the state reward of the current state is accumulated to a single reward. This approach implies that the optimal solution is a sequence of states and puts emphasis on the whole transition path.[3]

2) State Reward Specification and Utility Functions: In our approach, all state rewards are derived from clustering log measurements of similar past conditions, where the similarity is defined according to the external load. More specifically, we take the approach in [2] as a baseline: in each elasticity step where we instantiate a model, we group log measurements for a specific number of active VMs by their incoming load λ, and then those measurements are fed into a k-means clusterer, which returns k center points. The center of the biggest log measurements' cluster is the one which is selected as the most representative point for every state. However, when reaching such a state as a result of an elasticity action, the real state encountered may be closer to one of the remaining center points. To overcome this concern, we can extend the model so that it explicitly covers all the returned k centers with probabilities that are proportional to the size of their clusters; this requires the modeling option in Figure 1(right). Based on the extended model, we can define one state for each of the k clusters of log measurements. In this work, k is set to 3.

The next step is to map each such cluster to a reward, and this is achieved through a user-defined utility function. To comply with the rest of the model, utility functions are functions of the number of active VMs and are used to derive the state rewards in each model instantiation. Numerous utility functions can be used in elasticity scenarios, see [2] for a few of them. In this work, we want to focus on a utility function that penalizes both under- and over-provisioning. Since our example scenario considers NoSQL data stores, *thr* (for throughput) and *lat* (for latency) variables are two of the most significant properties to quantify performance. These two metrics are actually correlated: when *lat* is kept low, i.e., user requests are answered in time that does not exceed a given threshold, then *thr* can closely follow λ. The current work examines the following utility function (although the methodology presented above is independent of specific utility functions):

$$r = \begin{cases} 1 + 1/vms\_num & \text{if } lat \leq x \\ 0 & \text{if } lat > x. \end{cases}$$

In this function, x is the user-specified latency threshold, which should not be exceeded. The above utility function addresses the need for both under-provisioning and over-provisioning. Given that latency violations are due to under-provisioning, r penalizes the fact of activating fewer VMs than those needed. When the system is not under-provisioned, by placing the number of VMs to the denominator, we penalize over-provisioning. The model is extensible and it can accommodate additional variables (e.g., CPU utilization) and utility functions.

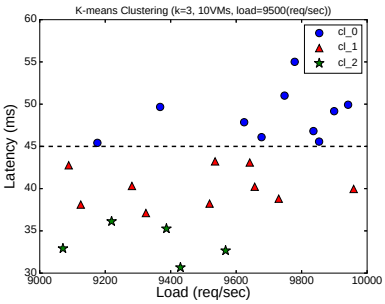


Figure 2: Clustering Detailed

Interestingly, the specification of model state rewards can be further refined taking into account the specific utility function employed. For example, instead of running k-means to all log measurements in order to derive the log clusters, one could map all log measurements that contain latency violations to a single cluster and then run k-means for $k-1$ clusters on the rest of the log measurements. The transition probabilities to each of those clusters for a specific action would be equal to the number of log measurements contained in that cluster. This type of model states makes a clearer distinction between desirable and undesirable states and despite of being utility function-specific, it facilitates to probabilistically guarantee under-provisioning, as explained later. A more detailed example is shown in Figure 2, where the current load is 9.5K, the number of VMs is 10 and the latency threshold is 45ms; the log entries close to that value are clustered in three groups, one for each of the three model states corresponding to 10VMs.

3) Prediction: As explained above, the state rewards are dynamically computed in each elasticity step according to the existing log measurements for the current external load λ. It is straightforward to extend our model states with information about the time step expected to visit each state (no diagrams are illustrated due to space limits) and then,

[3]Cumulative rewards without a discount factor cannot be used unless the number of steps is bounded; otherwise the maximum reward goes to infinity.

define the reward of model states according to the *predicted* external load at that time step. This is beneficial when the load is rapidly evolving, and as such, the actual reward of the same model state may be highly different at different time points. An ARIMA-based predictor can be built to predict future load values, as in [15]. Note that such a modeling flavour can also account for the delays in enacting an elasticity decision. For example, when adding a VM from a given state, the time step information of the resulting model state will consider the time overhead incurred to add a new VM and that VM to become fully operational (i.e., to receive data in a NoSQL cluster).

4) Complete Decision Policy Specification and Quantitative Analysis: We propose two decision policies that are both formalized (as being based to a formal MDP model) and dependable (as being the result of online verification). Probabilistic Computation Tree Logic (PCTL), encapsulated in the PRISM tool, allows for probabilistic quantification of described properties. The proposed policies differ in the way PCTL is employed as described below[4]:

- *SIMPLE (direct MDP solution,* BBA*, Instantaneous rewards):* With the help of PCTL, for each state of an instantiated model, we extract the maximum expected instantaneous reward for reaching that state, which is based on a direct solution of the MDP. Then, we choose the most profitable state, i.e., the whole problem is a max-max one. Finally, we either try to move the system to that state, or if this is not possible, e.g., if the amount of VMs required to be added exceeds the user-specified bounds of additions in a single step, we select the action that leads to the closest state to the target state.
- *ADVANCED (direct MDP solution,* BBS*, Cumulative rewards, probabilistic guarantees):* In this decision policy, we employ cumulative rewards. Then we ask for the maximum expected reward after a specified number of steps l. The result of the model verification process is a graph of possible sequences of actions. We collect all the first actions (i.e. the actions that begin from the current state) and using a second PCTL expression, we detect the one which has the minimum expected probability to lead to a latency violation, i.e., to visit a state with 0 reward according to the utility function presented earlier. This policy behaves better when we follow the modeling approach where for each number of active VMs, there is a single cluster that exclusively covers all violating log measurements (termed as *VC*).

Using PCTL formulae, the users can input additional high-level queries about the probability of the amount of additional resource metrics taking into consideration applied actions and reached states. For example, we can pose questions like the following: *"What is the maximum probability of the latency to be less than 30 milliseconds after state s_7 is reached?"*, which, in PRISM, can be formulated in this way: $Pmax =? [F\ latency < 30\ \&\ vms\_num = 7]$, where F implies the satisfaction of the reachability property [9]. Another example question is: *What is the probability that the system will remain in the decided state (assuming that the current environmental conditions do not change)?* Similarly, we can ask about minimum probabilities and any other metrics used in the model (e.g., throughput). In summary, we can pose any query involving maximum and minimum probabilities and/or rewards. In this way, the user can be more informed about the reason the selected decision was taken and has the ability to examine any metrics of the system, provided that they are employed in the utility functions and thus are captured by the model.

[4]An elasticity policy very close to SIMPLE has been proposed in [11] as *MDP2* along with additional flavours that we omit here for brevity; *ADVANCED* is novel.

III. Incorporation Into Existing Systems

Our elasticity decision approach can be encapsulated in every elastic manager provided that the latter meets the following requirements: it is capable (i) of collecting log measurements that are used for the training and the instantiation of the models and (ii) of enforcing the elasticity decisions taken.

In our prototype implementation, our PRISM-based decision technique is incorporated within Tiramola,[5] which is a modular, cloud-enabled, open-source system that enables elastic scaling of NoSQL clusters according to user-defined policies and incoming load. It allows seamless interaction with multiple IaaS platforms, requesting/releasing VM resources and orchestrating them inside a NoSQL cluster. Our approach is also compatible with cloud managers like the ones used by Amazon. In that case, the log measurements are provided through Amazon's EC2 *CloudWatch* and the decisions can be enforced through Amazon's EC2 *Auto Scaling* service. Note that the main current elasticity policy of Amazon is rule-based; in the next section, we compare the efficiency of rule-based decision policies against ours.

IV. Evaluation of Decision Policies

The main purpose of this section is to assess the efficiency of the decision policies enabled by our approaches. Since there can be too many combinations of models and decision policy configurations, we compare only a representative subset of decision policies:

- *RE*, which aims to reproduce pure reactive rule-based decision policies, where elasticity actions are triggered by constraint violations, like those enabled by Amazon EC2.
- *RL*, which employs the model in Figure 1(left), the Q-learning reinforcement learning approach and the state's reward is computed according to the center of the

[5]Publicly available from http://code.google.com/p/tiramola/

biggest cluster of log measurements, (thus reproducing the approach in [2], which represents the state-of-the-art in NoSQL elasticity).
- *SIMPLE*, which employs the model in Figure 1 with 3 model states per number of active VMs unless stated otherwise and a direct MDP solver is used through PRISM.
- *ADVANCED*, which employs the model in Figure 1 with 3 model states per number of active VMs unless stated otherwise and provides probabilistic guarantees as explained earlier. The number of steps l is set to 3.

A. Experimental Setup

We first collected logs from a real infrastructure, and then we ran emulated experiments based on those logs. The reason behind this is to allow for a completely fair comparison between the various techniques. In order to collect real data, we conducted log measurement experiments using the okeanos IaaS infrastructure [16], and the YCSB benchmark. For our NoSQL cluster, we have used 4 client VMs as load generators with 2 VCPUs and 4GB of RAM and 5GB storage each, and up to 18 cassandra server VMs (minimum 8 VMs) with 2 VCPUs, 2GB of RAM and 20GB storage each. The server VMs were created and booted before the experimental procedure. In all cases, the OS was "Debian Base" (7.4) running Linux 3.12.6 kernel the java VM runtime from the SUN JRE v1.7.0_51, and ganlia monitor v3.3.8 [17]. Cassandra v2.0.9 with 256 virtual nodes per host and a replication factor of one is installed and configured on every server VM. A (heavily modified) version of YCSB-0.1.4 ran on every client VM to produce the load; the modifications were made to support database metrics reporting on ganglia.

The workload consists of asynchronous read requests in uniform distribution. We have created varying sinusoidal load from 4000 (req/sec) up to 16000 (req/sec). We collected measurements every 30 secs, and in each sine period, there were 360 measurements.

The collected measurements are used firstly, to populate the initial logs of each policy, and secondly, to emulate a real situation. Through emulation, we managed to fairly test each policy on an equal basis, which could not be done if each policy ran separately in a real cluster. In our emulation, a time unit corresponds to the measurement collection period, i.e., 30 secs. We allow an elasticity action to take place every 10 time units, to emulate a system that may modify the VMs every 5 mins (or 10 mins is cases of add action, to allow the system to stabilize). Later, we examine more frequent options and we allow for decisions every 30secs. As the emulated load is generated based on the logs, which also act as training set, we consider that the system is well trained; note that some policies like *RL* are more sensitive than others to the quality of training [13].

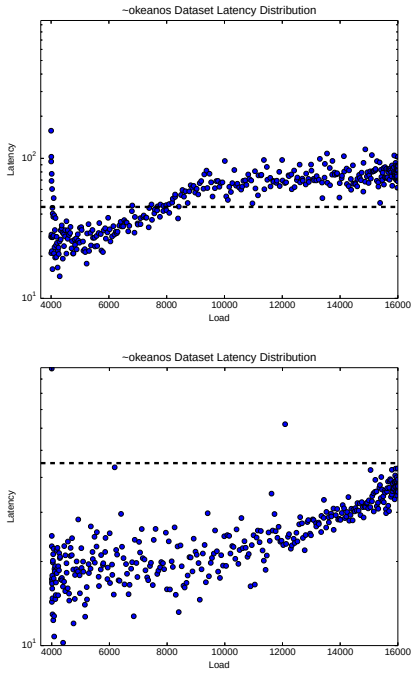


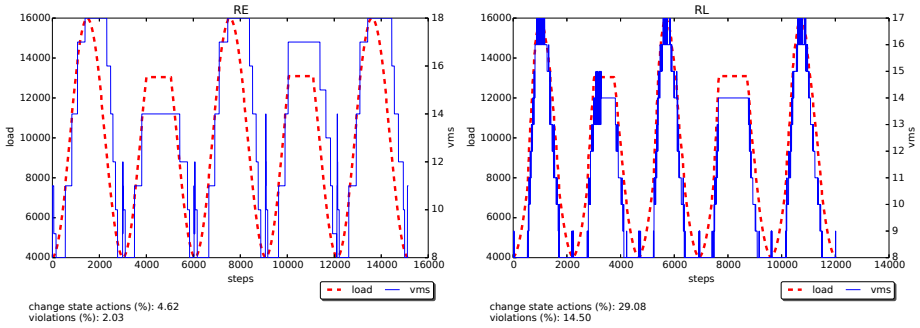
Figure 3: Latencies for 8 (top) and 18 (bottom) VMs

The load applied during elasticity experiments is a 5 period sinusoidal workload varying from 4000 (req/sec) to 16000 (req/sec) coupled with with 2 plateau periods at 13000 (req/sec) for 1000 time units each. In every up-scale action, up to 3 VMs can be added, while during down-scaling, up to 2 VMs are allowed to be removed in a single step.

The latency threshold in the utility function is set to 45ms; later, we examine further thresholds. Additionally, for *RE*, a lower latency threshold is set to 20 ms to trigger a remove action. Figure 3 presents the latency distribution in two characteristic states of the collected dataset, where the dotted line shows the latency threshold. For the lowest amount of load, there are few latency values that violate the threshold (mostly caused by the cold cache of the system at the beginning of the measurement collection) and the system can handle load up to about 8000 req/sec. For the maximum number of active VMs (18), except from a few outlier measurements, the system can handle the full amount of the incoming load.

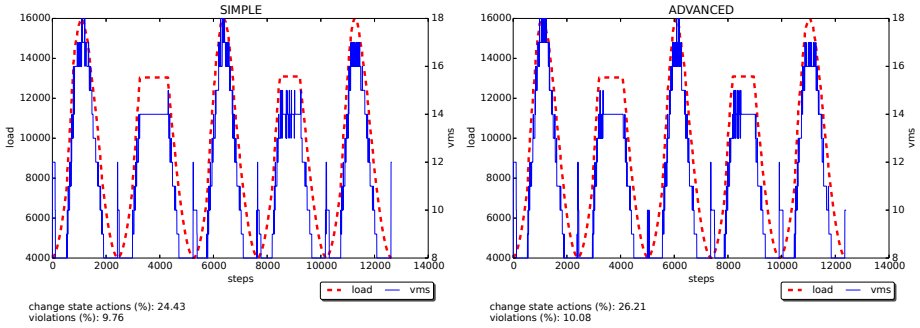
B. Experimental Results

In Figure 4, we present the adaptation of the number of VMs to the incoming load for each policy. All the policies can broadly follow the load variation, however *RE* (Figure 4a) does that in a less close fashion, as it takes the lowest number of state change actions (additions/removals) (4.62% of the total actions). This helps in avoiding under-provisioning and it experiences violations in only 2% violations of the steps; however it increases over-provisioning as will be discussed later. The other policies



(a) *RE*

(b) *RL-MB*



(c) *SIMPLE*

(d) *ADVANCED*

Figure 4: Variation of the external load and the number of active VMs

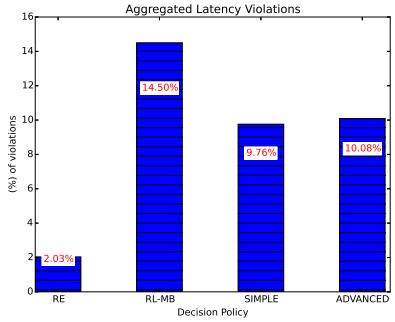


Figure 5: Aggregated Latency Violations

Decision Policy	*RE*	*RL-MB*	*SIMPLE*	*ADVANCED*
avg. utility	0.35	0.64	0.61	0.63

Table I: Average normalized utility

apply more state change actions to cope with the constantly evolving load change: *RL*(Figure 4b) performs changes in 29.08% of the steps, *SIMPLE*(Figure 4c) in 24.43% , and *ADVANCED*(Figure 4d) in 26.21%. The instability on the minimum amount of load (4000 (req/sec)) in every decision policy is explained by the outlier values presented in Figure 3.

We quantify under-provisioning by counting the number of steps where a latency violation occurs, as shown in Figure 5. *RE* policy seems to achieve the best behaviour regarding avoiding under-provisioning as the percentage of violations is 2.03%, while *RL* achieves the worst with 14.5% violations, i.e., it suffers from under-provisioning. The other two policies *SIMPLE* and *ADVANCED* exhibit an almost equal amount of latency violations, that is 9.76% and 10.08% respectively. For these numbers, we excluded the time steps in which the system tries to stabilize after an add action, but the pattern does not change if those time steps are considered.

Over-provisioning is quantified with the help of the utility function that penalizes the unnecessary use of extra machines. According to Table I, *RL*, *SIMPLE* and *ADVANCED* policies achieve significantly higher average utility than *RE*. The values in the table are normalized to the range [0,1]. The actual values range from 1+(1/18) = 1.056 to 1+(1/8) = 1.125. Combining the values in that table and in Figure 5, we see that *SIMPLE* and *ADVANCED* can strike a better balance between under-provisioning and over-provisioning.

Next we explain how the model extensions discussed in Sections II-B2 and II-B3 further improve *ADVANCED*. These are (i) to employ a utility function-specific MDP model state through *VC*, as presented in Section II-B2 and (ii) to employ a prediction module (*PRE*). Figure 6 examines the following combinations: (i) *ADV+VC* for *ADVANCED* combined with a model state exclusively for violations, and (ii) *ADV+VC+PRE* for additionally incorporating prediction of future external load values. The prediction error bound

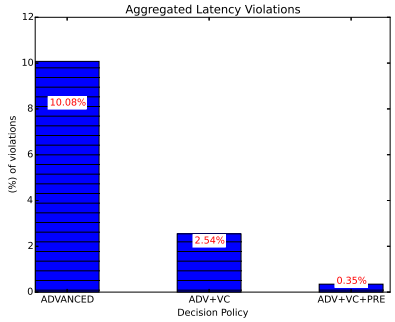


Figure 6: Aggregated Latency Violations for *ADVANCED* enhancements

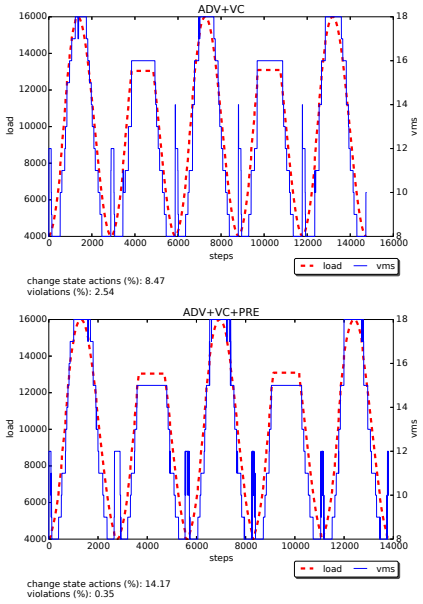


Figure 7: Variation of the external load and the number of active VMs for *ADV+VC* (top) and *ADV+VC+PRE* (bottom)

for the k^{th} future step is $0.05k$ times the minimum load. Both enhancements lead to a great decrease in the occasions of under-provisioning, which are nearly eliminated for *ADV+VC+PRE*. However, this comes at the expense of lower normalized utility (0.45). Figure 7 shows the behaviour of those decision policy alternatives. We have also examined the combination of the *SIMPLE* decision policy with a model state corresponding to a log measurement cluster with violations: the percentage of latency violations drops to 2.41% from 9.76%, but the average utility drops similarly to *ADV+VC+PRE*.

A final note is that, in this setting, it is not surprising that *RL* yields the highest utility, because *RL* is based on an optimally solved MDP and we assumed full and accurate training of the Q-learning approach. However, *RL* cannot reach the level of under-provisioning avoidance that *SIMPLE* and *ADVANCED* do, which is attributed to the modeling extensions presented and the probabilistic guarantees.

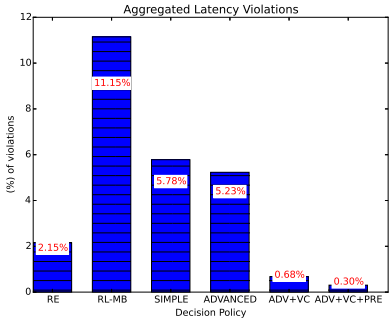


Figure 8: Results for higher decision frequency

Decision Policy	*RE*	*RL-MB*	*SIMPLE*	*ADVANCED*
avg. utility	0.34	0.71	0.61	0.61

Table II: Average normalized utility for higher decision frequency

C. Sensitivity Analysis

The policies and the evaluation setting described previously contain several fixed parameters. In this part of the experiments, we aim to provide string insights into how changing those parameters affects the results. We focus on (i) the decision frequency, (ii) the prediction accuracy, and (iii) the latency threshold.

1) Decision Frequency: In our experiments so far, decision making is activated every 10 time units or 5 emulated minutes, i.e., even if no action is decided at a specific point, the next consideration takes place after 10 time units. We now allow for making decisions at each time step (unless this step falls into a stabilization period). Figure 8 shows the behaviour of each policy and, with the help of Figures 5 and 6, we can see that our two proposals benefit the most in terms of the latency violations. For the example, the percentage of violations for the *ADVANCED* policy drops by 48%. This comes at the expense of a slight decrease in normalized utility. As shown in Table II, the utility of *RL* is increased by 10%, but the decrease in latency violations is only 23%. Figure 8 also considers the *ADVANCED* extensions. For *ADV+VC* policy, the percentage of violations drops by 73% (from 2.54% to 0.68%) and the average normalized utility is slightly increased (from 0.46 to 0.47). The *ADV+VC+PRE* policy is less affected by the change in the frequency of the decisions. Overall, our policies exhibit efficient elastic behaviour at both high and low decision making frequencies.

2) Prediction Accuracy: Here we assess the impact of the accuracy of the prediction. As we have already mentioned, the prediction error bound for the k^{th} future step is $0.05k$ times the minimum load. In this section we present experimental results for both accurate and more inaccurate predictions. For the latter, the error bound is doubled. Figure 9 summarizes the latency violations for the *ADV+VC+PRE* policy. The difference from the accurate setting is small.

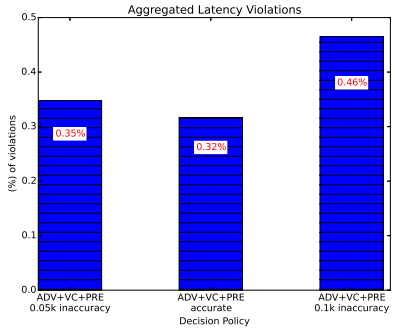


Figure 9: Results for different prediction accuracies

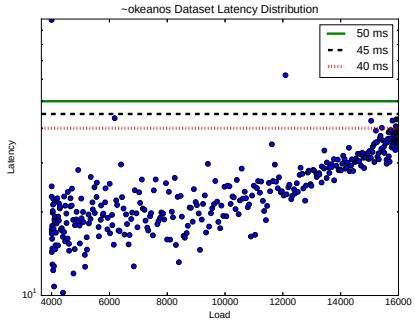


Figure 10: Different Latency Thresholds for 18 VMs

For even more inaccurate setting, the violations increase, but they are still less than 0.5%. This implies that our proposals are robust to prediction inaccuracies.

3) Latency Threshold: Until now, the latency threshold is set to 45 ms based on the latency distribution of the dataset for the maximum amount of active VMs (18). As Figure 3 (bottom) depicts, this is the minimum possible threshold to allow the maximum number of VMs to operate without violations. In this section we conduct experiments with lower and higher latency thresholds, namely 40 ms and 50 ms, respectively; see Figure 10, where we can observe that for the lower threshold (40 ms, red dotted line), even the maximum number of VMs is not capable of fully avoiding violations. Due to space constraints, we do not present detailed results. However, the same trend as in the previous cases appears: *ADVANCED* strikes a better balance between latency violations and utility, and its extensions further improve on this trade-off.

D. Decision Making Overhead

Using an Intel i7 4700M CPU (4 cores, 8 threads) with 8GM RAM, on average, the *RL* decision policy takes 0.013 secs to reach a decision, while our most expensive policy that invokes PRISM, namely *ADV+VC+PRE*, takes 5.8 secs. *RE* decides almost instantly (0.0002 secs). However, the difference of two orders of magnitude in the running time between the *RL* and our proposals is insignificant in practice, where we typically take elasticity actions every 5 or 10 mins.

V. Related Work

Our proposal is, to the best of our knowledge, the first advocator of online quantitative verification to drive cloud elasticity. In this section, we mention other representative approaches to the same problem and we discuss further differences with our proposal.

Using MDPs combined with reinforcement learning-based policies to decide the number of VMs has been proposed in [2], [13]. Compared to those proposals, we allow for direct MDP solvers, dynamically instantiated models and quantitative analysis.

The authors in [18] combine cloud elasticity with anomaly prevention. This proposal utilizes a prediction technique based on system metrics to vertically scale the resources of the VMs or to decide for VM migration, i.e., they consider different forms of elasticity, as is also the case in [4], [5]. In our work we employ prediction; however, analyzing the efficiency and effectiveness of prediction techniques is an interesting direction that we leave for future work. Complementarily to us, [19], [15] deal with heterogeneity issues, while we assume that all VMs are of the same type.

A significant number of proposals use rule-based techniques to guide the elasticity, e.g., [20], [21]. The former presents an enhanced rule-based technique with predictive capabilities. In [21], a technique is proposed that addresses the implications of an elastic action across multiple dimensions, providing for example the cost implication of a horizontal scaling action. None of those techniques is accompanied by online probabilistic verification of elasticity properties. For elasticity in cloud data stores, there are several proposals, such as [22], [23], [6], [3]. Apart from being limited to a specific setting, they tend to focus on satisfaction of strict SLOs, instead of maximizing utility.

Finally, model checking and runtime quantitative verification for cloud solutions other than horizontal scaling has been proposed in [24] and [25]. The former, utilizes PRISM to guide service adaptation, while the latter presents a technique to predict the minimum cost of cloud deployments using PCTL over MDP models.

VI. Summary and Future Work

This work presented a formal, probabilistic model checking-based approach to resizing an application cluster of VMs so that elasticity decisions are amenable to quantitative analysis. We presented MDP elasticity models and associated elasticity policies that rely on the dynamic instantiation of such models. We also conducted experiments using real datasets, and we presented results showing that we can significantly decrease the frequency of user-defined threshold violations and attain high utility values; these aspects are directly related to under-provisioning and over-provisioning, respectively.

In this work we have shown but not fully exploited the potential of MDP models, which we plan to do in the

future. MDP models can naturally capture complementary non-deterministic aspects of elasticity in real systems, such as provision for failure or long delays to enforce an elasticity decision and support for additional forms of elasticity like vertical resizing (e.g. resizing of CPU,RAM resources) and/or taking into consideration different VM types. Other directions for future work include the consideration of additional utility functions that are more directly associated with common usage and charging policies on clouds (e.g., to consider the charging-by-hour model to reason about over-provisioning), mitigating the impact of outdated log measurements in cases where there are significant shifts in the system behaviour, and more thorough experimental analysis.

Acknowledgments: This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund."

REFERENCES

[1] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in *Proc. of ICAC'13*, 2013, pp. 23–27.

[2] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, elastic resource provisioning for nosql clusters using tiramola," in *CCGrid'13*, 2013, pp. 34–41.

[3] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, p. 14, 2012.

[4] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic resource scaling for multi-tenant cloud systems," in *SOCC*, 2011, pp. 5:1–5:14.

[5] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *CNSM*, 2010, pp. 9–16.

[6] B. Trushkowsky, P. Bod?k, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "The SCADS director: Scaling a distributed storage system under stringent performance requirements." in *FAST*, 2011, pp. 163–176.

[7] J. O. Kephart and D. M. Chess, "The vision of autonomic computing." *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[8] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69–77, 2012.

[9] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker, "Automated verification techniques for probabilistic systems," in *Formal Methods for Eternal Networked Software Systems (SFM'11)*, 2011, pp. 53–113.

[10] M. Kwiatkowska, G. Norman, and D. Parker, "Prism: probabilistic model checking for performance and reliability analysis," *SIGMETRICS*, vol. 36, no. 4, pp. 40–45, 2009.

[11] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas, "Cloud elasticity using probabilistic model checking," *CoRR*, vol. abs/1405.4699, 2014.

[12] J. Kuhlenkamp, M. Klems, and O. Röss, "Benchmarking scalability and elasticity of distributed database systems," 2014, pp. 1219–1230.

[13] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck, "From data center resource allocation to control theory and back," in *IEEE CLOUD*, 2010, pp. 410–417.

[14] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.

[15] Q. Zhang, M. F. Zhani, R. Boutaba, and J. L. Hellerstein, "Harmony: Dynamic heterogeneity-aware resource provisioning in the cloud," in *ICDCS*, 2013, pp. 510–519.

[16] V. Koukis, C. Venetsanopoulos, and N. Koziris, "∼okeanos: Building a cloud, cluster by cluster," *IEEE Internet Computing*, vol. 17, no. 3, pp. 67–71, 2013.

[17] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[18] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *ICDCS*, 2012, pp. 285–294.

[19] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *Proc. of the IEEE Int. Conf. on Cloud Engineering (IC2E)*, 2014.

[20] L. Moore, K. Bean, and T. Ellahi, "A coordinated reactive and predictive approach to cloud elasticity," in *CLOUD COMPUTING 2013*, 2013, pp. 87–92.

[21] G. Copil, D. Moldovan, H. L. Truong, and S. Dustdar, "Multi-level elasticity control of cloud services," in *ICSOC*, 2013, pp. 429–436.

[22] A. Al-Shishtawy and V. Vlassov, "Elastman: elasticity manager for elastic key-value stores in the cloud," in *CAC*, 2013, p. 7.

[23] H. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," in *ICAC*, 2010, pp. 1–10.

[24] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic qos management and optimization in service-based systems," *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 387–409, 2011.

[25] D. Perez-Palacin, R. Calinescu, and J. Merseguer, "Log2cloud: Log-based prediction of cost-performance trade-offs for cloud deployments," in *ACM SAC*, 2013, pp. 397–404.