Performance Modelling and Verification of Cloud-based Auto-Scaling Policies

Alexandros Evangelidis, David Parker, Rami Bahsoon
School of Computer Science
University of Birmingham
Birmingham, United Kingdom
E-mail: {a.evangelidis, d.a.parker, r.bahsoon}@cs.bham.ac.uk

***Abstract*—Auto-scaling, a key property of cloud computing, allows application owners to acquire and release resources on demand. However, the shared environment, along with the exponentially large configuration space of available parameters, makes configuration of auto-scaling policies a challenging task. In particular, it is difficult to quantify, a priori, the impact of a policy on Quality of Service (QoS) provision. To address this problem, we propose a novel approach based on performance modelling and formal verification to produce performance guarantees on particular rule-based auto-scaling policies. We demonstrate the usefulness and efficiency of our model through a detailed validation process on the Amazon EC2 cloud, using two types of load patterns. Our experimental results show that it can be very effective in helping a cloud application owner configure an auto-scaling policy in order to minimise the QoS violations.**

I. Introduction

Cloud computing has become the most prominent way of delivering software solutions, and more and more software vendors are deploying their applications in the public cloud. In cloud computing, one of the key differentiating factors between successful and unsuccessful application providers is the ability to provide performance guarantees to customers, which allows violations in performance metrics such as CPU utilisation to be avoided.

In order to achieve this, cloud application providers use one of the key features of cloud computing: *auto-scaling*, which allows resources to be acquired and released on demand. While auto-scaling is an extremely valuable feature for software providers, specifying an *auto-scaling policy* which can guarantee that no performance violations will occur is an extremely hard task, and "doomed to fail" [1] unless considerable care is taken. In addition, in order for a *rule-based* auto-scaling policy to be configured correctly, there has to be an in-depth level of knowledge from the application manager, which is not necessarily true in practice [2]. This is the case even when a single auto-scaling rule needs to be specified.

Furthermore, public cloud providers such as Amazon EC2 and Microsoft Azure have started to offer more sophisticated rule-based policies to their users, which allow them to combine a set of auto-scaling rules. This type of policy is often referred to as *step adjustment*, and is used to specify a *fully automated* scaling policy. However, this "freedom" of being able to specify multiple auto-scaling rules comes with the cost of an extremely large configuration space. In fact, it is exponential in the number of performance metrics and predicates, making it virtually impossible to find the optimal values for the auto-scaling variables [3]. In addition, methods such as *trial-and-error* while the application is running are immediately ruled out, since it could have detrimental effects for the application owner to let the auto-scaler control the application with a complex policy whose effects on the Quality of Service (QoS) are unknown.

As noted in [4], auto-scaling policies "tend to lack correctness guarantees". The ability to specify auto-scaling policies that can provide performance guarantees and reduce violations of Service Level Agreements (SLAs) is essential for more dependable and accountable cloud operations. However, this is a complex task due to: (i) the large configuration space of the conditions and parameters that need to be defined; (ii) the unpredictability of the cloud as an operating environment, due to its shared, elastic and on demand nature; and (iii) the heterogeneity in cloud resource provision, which makes it difficult to define reliable and universal auto-scaling policies. For example, looking at public cloud providers, one can observe that there is no guarantee on the time it will take for an auto-scale request to be served, nor whether the auto-scale request will receive a successful response or not.

In order to address the aforementioned challenges, we propose a novel approach based on performance modelling and *probabilistic verification*, which is a formal approach to generating guarantees about quantitative aspects of systems exhibiting probabilistic behaviour. In particular, we use *probabilistic model checking* and the PRISM tool [5], [6], where guarantees are expressed in quantitative extensions of temporal logic and numerical solution of probabilistic models is used to quantify these measures precisely.

We use this as a formal way of quantifying the uncertainty that exists in today's cloud-based systems and as a means of providing performance guarantees on auto-scaling policies for application designers and developers. Another important novel aspect of our approach is the combination of probabilistic model checking with *Receiver Operating Characteristic (ROC)* analysis during empirical validation. This allows us not only to refine our original probabilistic estimates after collating real data and to validate the accuracy of our model, but also to obtain global QoS violation thresholds for the policies.

The probabilistic model is a discrete-time Markov chain

(DTMC), which we specify in the modelling language of the PRISM tool. The initialisation phase of the model begins by gathering CPU utilisation and response time traces, which are then fed into a *k-means* clusterer, in order to represent the states of our model under the different outcomes of average CPU utilisation and response time. Additionally, our model accounts for the variability in time between different auto-scale requests by introducing a waiting time parameter, on the scale of 1 (best) to 5 (worst), which can be specified by the cloud application developers or administrators, to verify their auto-scaling policy under different scenarios. It also takes into account the stochasticity of auto-scale requests, by transitioning between the different waiting times with a probability p, which can be specified by the user a priori, or left as a free parameter in order to probabilistically verify an auto-scaling policy under different values of p.

We demonstrate the correctness and usefulness of this approach through an extensive validation, using real VMs running on the Amazon EC2 cloud, under two different load patterns.[1] In order to validate the accuracy of our model, we perform *Receiver Operating Characteristic (ROC)* analysis, which is widely used in machine learning and data mining [8]. In a sense, we follow a relatively similar approach to the validation of classification models, by treating the probability as a ranking measure which determines the likelihood of the event of interest, in our case the probability for a violation. ROC can be used to help the decision maker to select the appropriate classification threshold, by quantifying the trade-off between sensitivity and specificity. Additionally, it can be used to validate the accuracy of a classification model, by computing the AUC (Area Under Curve) metric, which is one of the most commonly used summary indices [9].

Our validation starts by ordering the probabilities that have been computed from our PRISM model for each auto-scaling policy. Then, we plot the respective ROC curve by computing the points for the respective thresholds [0..1]. Through this analysis, we are able to find the optimal threshold of discriminating between the auto-scaling policies that could result in a CPU/response time violation. Our criterion of optimality is the point which minimises the Euclidean distance between the ROC curve, and point (0,1), which is often called the point of "perfect classification". Also, this gives us the ability to refine our original violation estimates after we have seen the real data, and to obtain a global threshold for distinguishing between auto-scaling policies. After plotting the ROC curve, we compute the AUC, which in our case can be interpreted as the number of times our model can distinguish performance violations/non-violations of randomly selected auto-scaling policies. This is an "important statistical property" [8] of AUC, and one of the reasons that it has been used so widely for validating the performance of classifiers.

Our modelling and verification framework is intended to minimise the time and costs for cloud application owners who do not have the resources or the desire to clone their existing applications in order to test them in a cloud-based environment. It can also provide valuable assistance in designing, analysing and verifying the auto-scaling policies of applications and services which are being deployed on public clouds, and could help in providing robust performance guarantees.

Probabilistic model checking has previously been applied to a wide range of application domains, including aspects of cloud computing [4], [10]. However, to the best of our knowledge, this is the first work to use probabilistic model checking to provide performance guarantees for auto-scaling policies from the perspective of a cloud application provider. We believe it is also the first case where the results of probabilistic verification have been validated using VMs running on a major public cloud provider, Amazon EC2, and the first usage of ROC analysis to validate such results.

II. Preliminaries

A. Rule-based auto-scaling policies

An *auto-scaling policy* [3] defines the conditions under which capacity will be added or removed, in order to satisfy the objectives of the application owner. Auto-scaling is divided into scaling-up/-down, and scaling-out/-in methods, with the two methods also defined as *vertical* (add more RAM, CPU to existing VMs) and *horizontal* (add more "cheap" VMs) scaling. In this work, we focus on scaling-out and -in, since it is a commonly used and cost-effective approach.

The main auto-scaling method that is given to application providers by all public cloud providers today (e.g., Amazon, Microsoft Azure, Google Cloud) is *rule-based*. The rule-based method is the most popular, and is considered to be the *state-of-the-art* in auto-scaling an application in the cloud [11]. It has been widely adopted for its pragmatic reasons: on the surface, it is simple and intuitive to configure by administrators and application providers. Moreover, when employed in certain and repeatable contexts, the approach can be efficient and cost-effective. However, as the cloud exhibits dynamic and unpredictable workload usage, rule-based approaches can be ineffective and limited. Our research hopes to address these limitations.

In a rule-based method, the application provider has to specify an upper and/or a lower bound on a performance metric (e.g., CPU utilisation) along with the desired change in capacity for this situation. For example, a rule-based method that will trigger a scale-out decision when CPU utilisation exceeds 60% might take the form: *if cpu_utilisation* $> 60\%$ *then add* 1 *instance* [12]. The performance metrics that public cloud providers usually follow include CPU utilisation, throughput and queue length. In this work, we consider auto-scaling decisions based on CPU utilisation, as it is one of the most important metrics in capacity planning, and also the most widely used in auto-scaling policies. The auto-scale control panel of the Amazon EC2 cloud can be seen in Figure 1.

B. Probabilistic model checking and PRISM

PRISM [6] is a *probabilistic model checker*, which supports the construction and formal quantitative analysis of various

[1] We have made the data used to validate our model publicly available [7].

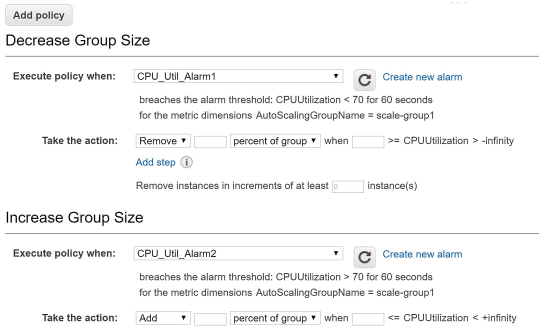

Fig. 1. Amazon EC2's auto-scale control panel.

probabilistic models, including discrete-time Markov chains (DTMCs), continuous-time Markov chains and Markov decision processes, which are expressed in PRISM's modelling language. A model in PRISM is broken down into *modules*, which represent different components of the system/process being modelled. The *state* of the model comprises values for a set of variables, which are either local to some module or global for the whole model.

Here, we use DTMCs, which are well suited to modelling systems whose states evolve probabilistically, but without any nondeterminism or external control. They are therefore appropriate here, where we want to verify auto-scaling policies, whose outcomes are probabilistic. Formally, a discrete-time Markov chain is defined as follows.

Definition 1. A *DTMC* is a tuple $M = \langle S,P,AP,L \rangle$where:

- S is a finite set of states;
- $\mathbf{P} : S \times S \rightarrow [0,1]$ is a transition probability matrix;
- AP is a finite set of atomic propositions;
- L: $S \rightarrow 2^{AP}$ is a labelling function.

For DTMCs, properties of the model are specified in PRISM using an extension [13] of the temporal logic PCTL (probabilistic computation tree logic) [14]. A typical property is $P_{\bowtie p}[\psi]$, where $\bowtie \in \{\leq, <, >, \geq\}$ and $p \in [0,1]$, which asserts that the probability of event ψ occurring meets the bound $\bowtie p$. As an example, in our model, where we want to verify whether the auto-scaling decisions will drive the cloud application to a state where the utilisation will be less than 95% with probability greater than 0.7, the following formula will be checked : $P_{>0.7}[F(util < 95)]$.

In addition, PRISM supports numerical properties such as $P_{=?}[F\ fail]$, which means "what is the probability for the cloud application to end up in a failed state, as a result of the auto-scaling decisions made?". Of course, what is considered a *failed* state will differ between cloud application owners, according to the relative importance they put on the non-functional aspects of their application. PRISM allows a wide range of such properties to be specified and analysed.

III. Formal modelling of rule-based auto-scaling policies

A. State representations

We model the dynamics of the auto-scaling process as a DTMC, the states of which are updated in each time step, corresponding to one minute. We have discretised time into one minute slots since this is an appropriate duration to gather and analyse the "external" data that constitute the state (e.g., arrivals, utilisation levels). In addition, due to the general uncertainty that manifests in a cloud environment, time intervals of less than one minute would make limited sense. They would result in unmanageably large spaces due to, for example, small temporal spikes in utilisation levels which can be ignored in our model. It would also be unrealistic to assume that a time step of less than one minute is long enough to flag a violation, and consequently to send an auto-scale request.

The states in our model can be divided into *deterministic* and *stochastic*. The former model the deterministic aspects of an auto-scaling policy that is to be verified. This is similar to a realistic case, where the cloud application will transition to states that have been previously defined by the application manager in the auto-scaling policy. To make the analogy more concrete with a realistic example, deterministic states in our model encode the subset of the conditions that apply to a cloud-based application, which determine whether an auto-scale request will be sent to the cloud provider or not. Stochastic states encode the probabilistic outcomes or responses of the auto-scale requests.

B. k-means clustering

Before the clustering process begins, we standardise the CPU utilisation and response time values by computing their *z-scores* [15]. Then, the model is initialised after *k-means* clustering has been run on the CPU utilisation, and response time traces. The value of k is also the number of different outcomes that could happen when a scale-out or a scale-in action occurs. Equivalently, k can be thought of as the number of states per number of VMs in operation.

As a result, as the size of k grows larger, the more detailed the possible state representations will be, possibly at the cost of adding a degree of overhead in the verification process. Conversely, for smaller values of k, the scalability of the verification process is improved, at the possible cost of representing the states in a "coarser" way. In our model, after experimenting with different cluster sizes, we have set $k = 5$.

C. Encoding auto-scaling policies in PRISM

We have developed a DTMC model, in the PRISM modelling language (≈1000 lines of code), that encapsulates the dynamics that affect the auto-scaling process, and can be used to formally verify auto-scaling policies. The "free" parameters (*constants*) which are left to the user of the model to set, are: i-iv) step adjustments for scale-out and scale-in rules, v) the increment which specifies the number of instances which will be added. vi) the decrement which specifies the number of

TABLE I
MODEL PARAMETERS

Model Parameters		
Time variables	*t*	Discretised time (step) for each state
	WAIT_TIME	Waiting time for an auto-scale request to be served
	MAX_TIME	Maximum time the model will run
Virtual machines	*INITIAL_VMs*	Number of reserved VMs
	cVMs	VMs operating currently
Auto-scale requests	*scale_out_1*	Scale-out request for CPU utilisation between 60% and 70%
	scale_out_2	Scale-out request for CPU utilisation between 70% and 100%
	scale_in_1	Scale-in request for CPU utilisation between 30% and 40%
	scale_in_2	Scale-in request for CPU utilisation between 0% and 30%
Other actions	*wait*	No action is taken
Cloud provider auto-scale responses	*satisfy_scale_out_request*	Adds the percentage of capacity requested
	satisfy_scale_in_request	Removes the percentage of capacity requested
Adjustment	*s_o_adj_1*	Percentage of capacity to be added when *scale_out_1* is chosen
	s_o_adj_2	Percentage of capacity to be added when *scale_out_2* is chosen
	s_i_adj_1	Percentage of capacity to be removed when *scale_in_1* is chosen
	s_i_adj_2	Percentage of capacity to be removed when *scale_in_2* is chosen
Boolean auto-scaling variables	*scale_out_trigger*	Coordinates scale-out requests
	scale_in_trigger	Coordinates scale-in requests
	capacity_added	Indicates that capacity has (not) been added
	capacity_removed	Indicates that capacity has (not) been removed
Performance metrics	*arrivals*	Arrivals (requests)
	served_reqs	Average requests/jobs served per 1 time unit
	util	Average VM CPU utilisation
	r_t	Average response time
Probabilities	*q*	Prob. of the different outcomes of *util* and *r_t* (based on k-means)
	p	Probability of delay in serving an auto-scale request
Increments / Decrements	*inc*	Scale-out policy adds instances in increments of *inc*
	dec	Scale-in policy removes instances in decrements of *dec*

TABLE II
AN EXAMPLE OF A STEP ADJUSTMENT AUTO-SCALING POLICY AS SEEN IN AMAZON'S DOCUMENTATION [16]

Scale-out policy			
Lower Bound	Upper Bound	Adjustment	Metric value
0	10	0	$50 \leq value < 60$
10	20	10	$60 \leq value < 70$
20	null	30	$70 \leq value < +\infty$
Scale-in policy			
Lower Bound	Upper Bound	Adjustment	Metric value
-10	0	0	$40 < value \leq 50$
-20	-10	-10	$30 < value \leq 40$
null	-20	-30	$-\infty < value \leq 30$

instances which will be removed, vii) the number of VMs that are currently reserved, viii) the maximum time the model will run the ix) probability p, and x) the time that it will take for an auto-scale request to be satisfied. In Table I we show the parameters that characterise our model, and in Listing 1 which of the parameters are expressed as *constants* in PRISM.

Listing 1
CONSTANTS IN PRISM

```
const double s_o_adj_1;
const double s_o_adj_2;
const double s_in_adj_1;
const double s_in_adj_2;
const inc; const dec;
const INITIAL_VMs;
const MAX_TIME;
const double p; const WAIT_TIME;
```

The first seven constants are under the control of the application provider, and represent the values that an application owner would have to set in reality. The last two represent parameters that are not controllable, and are being used as a basis for modelling and analysis of scenarios of interest (e.g., worst-case scenarios). After these parameters have been specified, the model transitions, with a probability q, to the possible outcomes of CPU utilisation and response time which are associated with the particular number of VMs. These outcomes, and the respective probability q, are set according to *k-means* which has been run beforehand. The verification process is explained in a later section.

Then, depending on the granularity of the auto-scaling policy, there can be multiple states each of which is a deterministic state. For example, for the step adjustment auto-scaling policy in Table II, the deterministic state would have had six different representations (one for each CPU utilisation interval). Then, depending on the utilisation level (similar to a realistic auto-scaling policy), the model transitions to the appropriate deterministic state, which causes the respective auto-scale request to be triggered or not, resulting in a transition to a stochastic state. This represents the fact that the auto-scaling request has been sent to the cloud provider, and is ready to be processed.

According to the type of scenario one wishes to analyse and verify, different transitions will occur. For example, let us assume that the probability p is set to 0.2, which means that with probability 0.2 the auto-scale request will be satisfied in the next time step, and with probability 0.8 the auto-

scale request will be satisfied according to the *WAIT_TIME* which is associated with p. Then, for the first case the model will transition with probability p to a state where the *capacity_added* variable will be set to true, and subsequently the auto-scaling request will be immediately satisfied, resulting in a change in the overall VM capacity.

On the contrary, with probability $1 - p$, the model will transition to a state where the auto-scale request will be satisfied based on the best-effort reservation policy, and the respective boolean auto-scaling variables will be set to true/false accordingly, representing the realistic case in which the auto-scaling actions are being locked, and the application is forced to wait until the auto-scale request is satisfied. For example, when a *scale_out* transition happens, the corresponding state variable *scale_out_trigger* is set to *true*, preventing further scale-out or scale-in requests from being triggered, as would have happened in a realistic setting. In Listing 2, we show a sample of the PRISM code which determines the transitions, based on the probability p, that was set in the initialisation phase. The lines of code before the $\rightarrow$ symbol, show the predicates that have to be satisfied for the transition to occur.

Listing 2
TRANSITION BASED ON THE VALUE OF p.

```
[choice] (...)&(t<MAX_TIME)&
(scale_out_trigger=true|..)&
(best_effort=false)&(imm_res=false)->
p:(imm_res'=true) + 1-p:(best_effort'=true);
```

Also, the waiting time which was initialised before the model started will now start to decrement, and until the duration of the waiting time has elapsed, the model will be prohibited from sending any other auto-scale requests. Again, this is according to realistic cloud controllers in practice, where the application owner is prohibited from sending any more auto-scale requests until the one that has been sent has been satisfied. In Listing 3, PRISM sample code for the *scale_out_1* transition is shown.

Listing 3
SCALE-OUT STEP 1 TRANSITION IN PRISM

```
[scale_out_1] (...) & (cVMs>=1)&
(scale_out_trigger=false)&
(scale_in_trigger=false)&
(best_effort=false)&
(t<MAX_TIME)&(util>=60 & util<70)->
(scale_out_trigger'=true)&
(t'=t+1)&(actual_util'=ceil(util));
```

Furthermore, while the waiting time is active, the model continues to generate load for the VMs to process. As has been described above, we abstract the generated load in the model, and our focus is directly on the impact of the load on the performance metrics (CPU utilisation, and response time), based on *k-means*. The transitions unfold in a similar manner, for example when the waiting time has completely elapsed, the auto-scale request will be satisfied, and the requested capacity will be added or removed.

IV. FORMAL VERIFICATION OF AUTO-SCALING POLICIES

The verification process consists of two phases. In the first phase, we generate load on the rented VMs to gather at least 100 data points for CPU utilisation and response time, for each VM number between 1 and 8. These 100 data points correspond to approximately 100 minutes of load generation, monitoring, and data gathering for each VM, and for each load type, resulting in a 26 hour data collection approximately. These data points are used for the initialisation of our model. In the second phase, we use *k-means* clustering to cluster the respective data points, which correspond to different outcomes of CPU utilisation and response time. Once the clustering process is finished, the clusters are fed into our model in PRISM, and once an auto-scaling policy or a set of auto-scaling policies is passed as an input to our model, we obtain the verification results.

TABLE III
AUTO-SCALING POLICIES FOR FORMAL VERIFICATION

Action	Inc/Dec	Min Util.	Max Util.	Initial VMs	Adjust.
Scale-out	[1..2]	60%	70%	2,3,4,8	[+10%]
Scale-out	[1..2]	70%	100%	2,3,4,8	[+30%]
Scale-in	[1]	0%	30%	2,3,4,8	[-30%]
Scale-in	[1]	30%	40%	2,3,4,8	[-10%]
Wait	-	40%	60%	2,3,4,8	0%

Throughout this process we obtain CPU utilisation and response time guarantees under two load patterns: "periodic" and "aggressive" (see Section V-B for details). Specifically, we are interested in computing the probability that the cloud application (consisting of all the VMs), will end up in a state where the utilisation is $\geq 95\%$, and the probability response time is ≥ 2 seconds for "periodic" load, and ≥ 5 seconds for the "aggressive" load (Listing 4) under the policies shown in Table III. We vary the *INITIAL_VMs* in the range [1..8] and *inc*, *dec* in the range [1..3].

Listing 4
PROPERTIES TO BE CHECKED

```
P=? [F util >= 95] //both load patterns
P=? [F r_t >= 2] // "periodic" load
P=? [F r_t >= 5] // "aggressive" load
```

In addition, since the outcome of the auto-scaling action depends on uncontrollable parameters, we vary the probability p, and the *WAIT_TIME* in our model. It is important to note that the verification process is not driven by too "optimistic" or too "pessimistic" parameter tuning. Specifically, we are interested in verifying for which set of "reasonable" variables in the auto-scaling policy, utilisation and response time guarantees

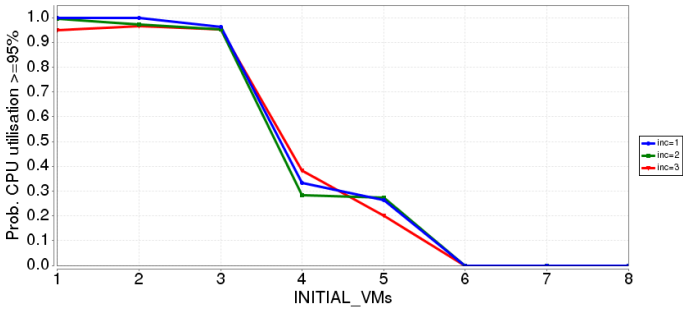

Fig. 2. PRISM results for P=? [F util $\geq$ 95] ("periodic" load)

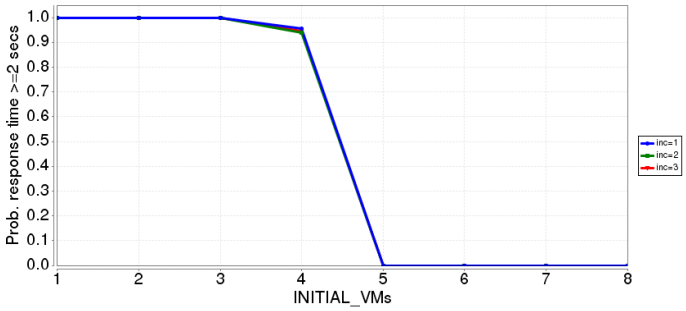

Fig. 3. PRISM results for P=? [F r_t $\geq$ 2] ("periodic" load)

hold. By "reasonable", we mean that we do not analyse auto-scaling policies under unrealistic conditions, such as choosing an increment of 20 VMs, since there is a non-negligible cost associated with renting the VMs. Also, we avoid unrealistic assumptions with respect to the time taken to satisfy auto-scale requests, by not varying p above 0.5. This fits our purpose of performing worst-case analysis as well. In our modelling and verification approach the *worst-case* is defined for the situation in which the auto-scale request will never be satisfied immediately ($p = 0$), and it would take at least 5 (*WAIT_TIME*$= 5$) time units to be satisfied. Conversely, the *best-case* is defined with $p = 0.5$ and *WAIT_TIME*$= 1$.

In Figures 2 and 3, we show the output of verification, for a specific auto-scaling policy, under "periodic" load. For the controllable parameters, we set the adjustments to $\pm 10\%$ and $\pm 30\%$ for the two steps, respectively, as in Table III. For space reasons, we omit the graph for the "aggressive" load. For the uncontrollable parameters, we set $p = 0.2$ and *WAIT_TIME*$= 3$, as an average case, which is not biased towards worst- or best-case scenarios. In Figure 2, we can observe that the probability for a CPU utilisation violation follows a decreasing trend as the number of reserved VMs increases. An important observation which we capture is the decrease by approximately 0.5 in the probability for a CPU utilisation violation when the cloud application starts serving HTTP requests with 4 VMs compared to 3. In Figure 3, the probability for response time violation fluctuates around 0.95, and drops sharply to 0 when the reserved VMs become 5.

V. Model Validation

The validation framework consists of three parts. The first is the experimentation setup on the Amazon EC2 cloud, the second is the load profile and the third is the ROC analysis. We describe each of these below. Data and other supplementary material from this process is available online [7].

A. Experimentation setup

The validation process uses a live public cloud setting: Amazon EC2. The architecture of this experimental setup, is similar to the one in Figure 4. We have created an auto-scaling group with minimum and maximum capacity of 1 and 8 VMs respectively. The VM types that were used were *t2.micro* with

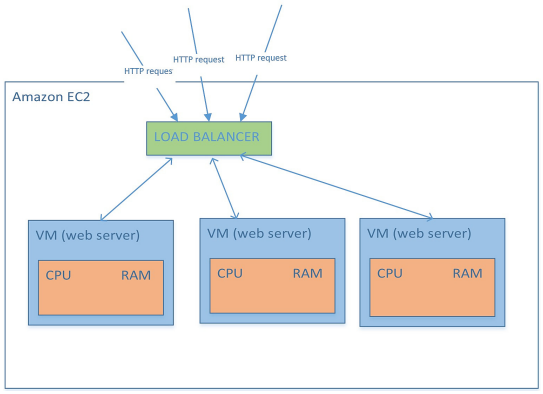

Fig. 4. Experimental setup on Amazon EC2

1 CPU and 1 GB of RAM. In order to simulate the auto-scaling process in the front layer (web servers) of a cloud-based application, a Python start-up script [17] was launched on those VMs, to simulate the HTTP processing and the CPU consumption. Specifically, the VMs were configured to process each request in 1 second and to send a 500 HTTP response code when 9 seconds have passed. Also, an Internet-facing load balancer was used which distributed the load in a round-robin fashion.

In addition, to monitor and log all the metrics of the auto-scaling group, we have used Amazons EC2's monitoring service, CloudWatch [18]. The performance metrics are averaged over all the VMs. We monitor and gather the performance metric data for each VM number and for each auto-scaling policy. For each of the policies shown in Table III, we generate load and monitor our VMs on the Amazon EC2 cloud for 10 minutes. Then, we repeat this process 30 times for each auto-scaling policy, resulting in 5 hours of data gathering per auto-scaling policy we are validating, approximately. These samples are then used to validate the verification results of our model.

B. Load profile

We generate two types of load in the VMs; a "periodic" and an "aggressive" load pattern. The main reason for choosing a periodic load pattern is because it is considered one of the most popular workload types in cloud computing [19]. To generate a periodic load we have used Apache JMeter [20],

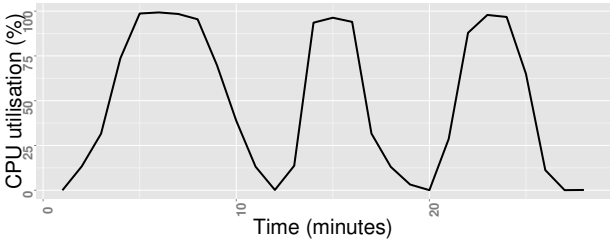

Fig. 5. Sample CPU utiilisation trace under "periodic" load.

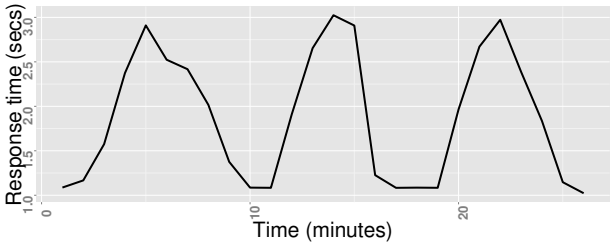

Fig. 6. Sample response time trace under "periodic" load

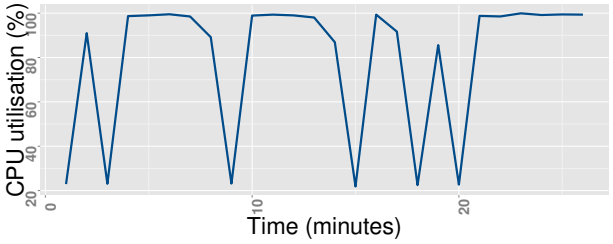

Fig. 7. Sample CPU utilisation trace under "aggressive" load.

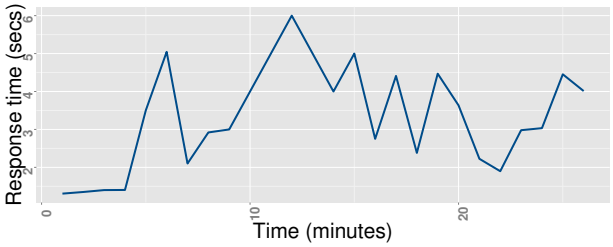

Fig. 8. Sample response time trace under "aggressive" load

which is a professional open source tool for testing web-based applications. Also, on top of Apache JMeter, we make use of the *Ultimate Thread Group* [21] extension for Apache JMeter, which gives us greater flexibility over the threads we are creating. Specifically, we create 3 *Ultimate* thread groups and, within each thread group, we start generating HTTP requests from 1 thread, and then we gradually increase the number of threads. Also, we keep the load duration of each thread for approximately 200 seconds.

The second type of workload we are considering has a greater degree of randomness, and our aim is to validate our model against an aggressive load pattern, but with an inherent degree of randomness. For this type of workload we make the assumption that HTTP requests arrive according to a Poisson process with exponentially distributed inter-arrival times.

In order to generate random variables to simulate the workload, the *inverse transform* sampling method was used, which is one of the most widely used sampling methods in performance modelling. This method is relatively simple once the CDF of the random variable X that is to be generated is known, and the CDF of X can be inverted easily, which holds in our case since the inter-arrival times of the HTTP requests in 1 time unit are exponentially distributed, and the inverse of the CDF of an exponential distribution has a closed-form expression. The algorithm works as follows [22]:

1) Generate $u \in U(0,1)$
2) Return $X = F_X^{-1}(u)$

As a result, step 2 will return a realisation of a random variable X from an exponential distribution. In our case, since we assume 1 time unit, we keep generating exponentially distributed random variables until their sum is 1. We run 1000 simulations and, after storing the results in a vector, we take a sample size of 50 instances.

In Figures 5–8 we show a sample of a CPU utilisation, and a response time trace under the two types of workload.

C. Results and model validation via ROC analysis

In this section, we give an overview of ROC analysis, and how it fits our purpose of discriminating between auto-scaling policies that could or could not result in a QoS violation. Our PRISM model takes as an input an auto-scaling policy x, and produces a continuous output which is a probabilistic estimate denoting the probability that an auto-scaling policy will result in a QoS violation/non-violation.

Effectively, we wish to find the mapping from x to a discrete variable $y \in \{0, 1\}$, with 0 and 1 indicating the non-violation, and violation cases, respectively. However, since the output of the model is continuous ($P(x = 1)$), and the prediction we want to make is binary, through ROC analysis we find a threshold t such that if $P(x = 1) \geq t$, then we predict that $y = 1$, and if $P(x = 1) < t$, then we predict $y = 0$. We choose t based on our optimality criterion which minimises the Euclidean distance between the ROC curve and point (0,1). Figures 9–12 show the ROC curves for CPU utilisation and response time under the two load patterns. In Figure 9, for example, the threshold t would be approximately 0.8.

However, it is important to note that what is considered optimal varies according to the problem one is trying to address, and the relative importance of missing, or increasing true and false positives. This threshold acts now as a global threshold, and based on that we compute the confusion matrix, and the associated performance measures, where the predicted outcome is determined by this threshold, and the actual values are obtained through real measurements on the Amazon EC2 cloud.

For the ROC curves in Figures 9–12, we plot the diagonal (red dashed line), which can be thought of as a baseline, which

would have been obtained by a random classifier, in order to show how the AUC (Area Under Curve) extends over the diagonal. AUC takes values between 0 and 1, with 1 indicating a "perfect" classifier. For example, if the AUC = 0.5, this is equivalent to a random classification model, and consequently the further the AUC extends over this diagonal, the greater the accuracy of the model.

For completeness, we also consider the MCC (Matthews Correlation Coefficient) [23], despite the fact this is often contrasted with AUC. MCC takes values between -1 and +1, indicating a negative and positive correlation between the predicted and the actual value. Below, we provide its definition, along with several other performance measures used to validate our model, in terms of the metrics from the confusion matrix.

- ACC = $\frac{(TP+TN)}{(TP+FP+FN+TN)}$(overall accuracy of model)
- TPR = $\frac{TP}{(TP+FN)}$ (true positive rate or sensitivity)
- TNR = $\frac{TN}{(FP+TN)}$ (true negative rate or specificity)
- FPR = $\frac{FP}{(FP+TN)}$ (false positive rate, 1 - specificity)
- FNR = $\frac{FN}{(FN+TP)}$ (false negative rate, 1 - sensitivity)
- MCC = $\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

In Tables IV and V, we show the results for these performance measures.

TABLE IV
PERFORMANCE MEASURES FOR "PERIODIC" LOAD

Perf. metrics	ACC	TPR	TNR	FPR	FNR	MCC
CPU util	0.92	0.88	1	0	0.12	0.83
Resp. time	0.9	0.82	0.94	0.06	0.18	0.77

TABLE V
PERFORMANCE MEASURES FOR "AGGRESSIVE" LOAD

Perf. metrics	ACC	TPR	TNR	FPR	FNR	MCC
CPU util	0.91	0.98	0.65	0.35	0.02	0.7
Resp. time	0.96	0.97	0.93	0.07	0.03	0.86

VI. Discussion of results

Our model captures accurately enough the probability of CPU utilisation and response time violations for the specific auto-scaling policies that were shown in the previous section. This can be seen from the AUC values for the auto-scaling policies under the two types of load, as shown in Figures 9–12. For auto-scaling policies which could result in CPU utilisation violation, the AUC value is larger under the "periodic" load, whereas for policies which could result in response time violations, the AUC is larger for the "aggressive" load. However, for both of these cases, the AUC is high (> 0.8), which shows the high accuracy of our model, under the thresholds [0..1], for the two types of workload.

In Tables IV and V we show the validation results of the model after picking a global threshold for each of the four cases. For the auto-scaling policies verified under the "periodic" load (Table IV), we note that the overall accuracy (ACC) is higher for CPU utilisation violation detection, compared to response time. An important observation is that TNR=1, which represents the fact that our model was able to detect, without any error, auto-scaling policies that would not result in CPU utilisation violation, and as a result there were no misclassifications of auto-scaling policies which did not cause a CPU utilisation violation in the 10 minute period. Moreover, TPR=0.88 indicates that 88% of the auto-scaling policies which did cause a CPU utilisation violation were correctly identified as such. For the response time, despite the fact that we see a marginal loss of 0.02 compared to the CPU utilisation in the overall accuracy of the model, we note that TPR and TNR achieve high values of 0.82 and 0.94, respectively. However, we note an increase in the FNR by 0.06.

For the auto-scaling policies validated under the "aggressive" workload (Table V), we note that the overall accuracy of the model with respect to CPU utilisation violation detection remains high (ACC=0.91). Furthermore, the increase in FPR (0.35), compared to the "periodic" load, means that our model flagged auto-scaling policies as CPU utilisation violators, when in fact they were not. One of the possible causes of this could have been the fact that our gathered CPU utilisation traces contained too many violations, compared to the initial gathered traces that were passed to *k-means*, in order to be used in the state representation of our model. This effect is due to the random nature of the load, and could potentially indicate that more samples are required.

Another observation is the very small value of FNR (0.02), which is considerably more important in our case, since the effects of not flagging an auto-scaling policy as a possible QoS violator could be more serious than the opposite scenario. Finally, for both types of workload, MCC is ≥ 0.7 for both CPU utilisation and response time, indicating a very strong positive relationship between the policies our model flagged as very likely to cause/not cause a violation, and the actual outcome when these policies were evaluated in the VMs in Amazon EC2 cloud. We consider the high value of MCC (≥ 0.7) as particularly important since MCC is a balanced measure of the quality of binary classifications, even in cases of imbalanced data.

VII. Related work

Probabilistic model checking has been employed with great success in recent years to verify and analyse properties of systems that manifest varying degree of uncertainty. Domains include automotive systems [24], security [25], biology [26] and many others. Lately, there has been an increased interest by researchers in applying probabilistic model checking to

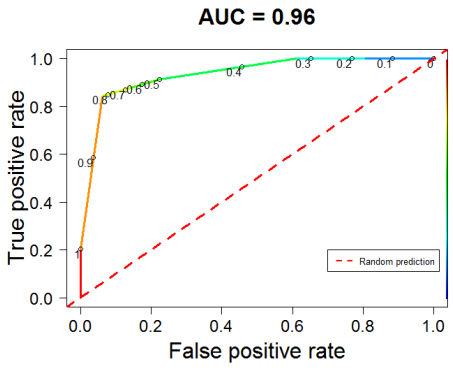

Fig. 9. ROC curve for CPU util. viol. ("periodic" load)

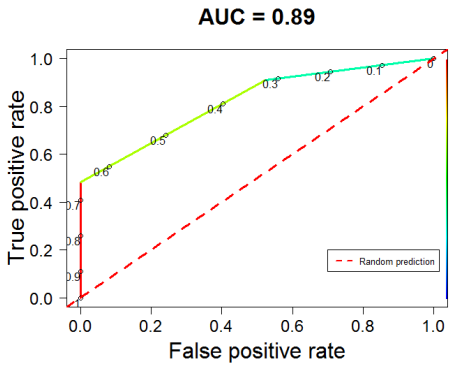

Fig. 10. ROC curve for resp. time viol. ("periodic" load)

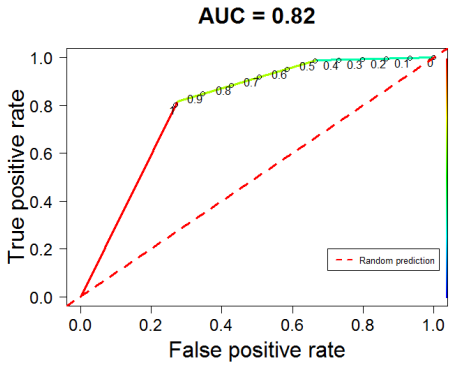

Fig. 11. ROC curve for CPU util. viol. ("aggressive" load)

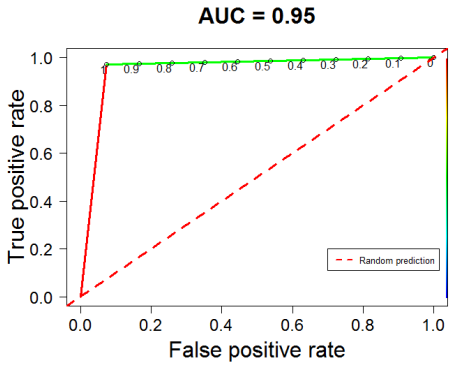

Fig. 12. ROC curve for resp. time viol. ("aggressive" load)

cloud computing. The main reason is that, from whichever perspective cloud computing is examined (e.g., IaaS, Paas, SaaS layer), there is an inherent degree of uncertainty, and there is a great need for this uncertainty to be quantitatively analysed.

Over the last five years, there has been an active interest from researchers in employing probabilistic model checking in dealing with the resource provisioning problems in the cloud. For instance, Fujitsu researchers [10] used probabilistic model checking to verify the performance of live migrations in the IaaS layer [10]. They assume that migration requests are distributed in a uniform way, which is not necessarily true in practice [27].

In addition, [4] proposed a Markov decision process model developed with PRISM, among other models, in order to formally verify different types of auto-scaling policies, including rule-based, and as a result their work is characterised by its breadth. The difference between our work and theirs is that we focus exclusively on rule-based auto-scaling policies, by developing one dedicated model (≈1000 lines of code) to simulate the dynamics of the auto-scaling process. As a result, we take a vertical in-depth approach in the auto-scaling process, by considering a significant number of parameters that occur in realistic cases.

Other work on the evaluation of auto-scaling policies, not using probabilistic model checking, includes [28], which proposed a performance metric for evaluating auto-scaling policies, but it is not clear where their experiments were conducted and to what extent their proposed metric can be helpful in a realistic cloud setting. The other differentiating factor in our work is that we perform an extensive validation of our model on a major public cloud provider (Amazon EC2). This means that we do not have, or assume, any type of control on the underlying auto-scaling mechanisms or on the VM provisioning methods and strategies employed by the cloud provider, and through our model, we are trying to infer the different outcomes that could happen. In general, however, we note that the use of probabilistic model checking for pragmatic use cases in the cloud is still at its infancy, and our research hopes to make an advancement towards bridging this gap.

VIII. Conclusion and Future Work

We presented a novel probabilistic verification scheme, followed by an extensive and robust validation on VMs rented on a major public cloud provider, Amazon EC2. This is, to the best of our knowledge, the first in-depth study of auto-scaling

policies, which is conducted on a public cloud provider, and not on a simulation toolkit or on a private cloud.

To this end, we have developed a Markov model using the PRISM model checker, in order to capture the dynamics of the auto-scaling process. This allowed us to compute probabilities of CPU utilisation and response time violation for each auto-scaling policy passed as an input to our model. Then, by using ROC analysis we were able to refine our original estimates, and find a global estimate which best represented a threshold for differentiating between auto-scaling policies which could be flagged as QoS violators or non-violators. Our experiments show that our verification scheme can be of valuable assistance to cloud application owners and system administrators in formally configuring, and verifying the auto-scaling policies of their applications/systems in the cloud.

In this work, we dealt mainly with the dynamics of an auto-scaling policy by varying the increments and the initial VMs in operation, as part of the controllable parameters. As future work, we will analyse the effects of varying the other controllable parameters of an auto-scaling policy, such as the percentages of the scale-out and scale-in adjustments. Finally, we will investigate further the temporal properties of the auto-scaling process such as the time the application spends in an overutilised versus an underutilised state.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck, "From Data Center Resource Allocation to Control Theory and Back," in *2010 IEEE 3rd International Conference on Cloud Computing*, Jul. 2010, pp. 410–417.

[2] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight Resource Scaling for Cloud Applications," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2012, pp. 644–651.

[3] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring Alternative Approaches to Implement an Elasticity Policy," in *2011 IEEE International Conference on Cloud Computing (CLOUD)*, Jul. 2011, pp. 716–723.

[4] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas, "Dependable Horizontal Scaling Based on Probabilistic Model Checking," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2015, pp. 31–40.

[5] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic Model Checking," in *Proceedings of the 7th International Conference on Formal Methods for Performance Evaluation*, ser. SFM'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 220–270.

[6] ——, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[7] "Suporting material," http://www.prismmodelchecker.org/files/ccgrid17/.

[8] T. Fawcett, "An Introduction to ROC Analysis," *Pattern Recogn. Lett.*, vol. 27, no. 8, pp. 861–874, Jun. 2006.

[9] W. J. Krzanowski and D. J. Hand, *ROC Curves for Continuous Data*, 1st ed. Chapman & Hall/CRC, 2009.

[10] S. Kikuchi and Y. Matsumoto, "Performance Modeling of Concurrent Live Migration Operations in Cloud Computing Systems Using PRISM Probabilistic Model Checker," in *2011 IEEE International Conference on Cloud Computing (CLOUD)*, Jul. 2011, pp. 49–56.

[11] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal Autoscaling in a IaaS Cloud," in *Proceedings of the 9th International Conference on Autonomic Computing*, ser. ICAC '12. New York, NY, USA: ACM, 2012, pp. 173–178.

[12] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically Scaling Applications in the Cloud," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, pp. 45–52, Jan. 2011.

[13] 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)*, ser. LNCS, M. Bernardo and V. Issarny, Eds., vol. 6659. Springer, 2011, pp. 53–113.

[14] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 1994.

[15] C. Romesburg, *Cluster Analysis for Researchers*. Lulu Press, 2004.

[16] "Dynamic Scaling - Auto Scaling," http://docs.aws.amazon.com/autoscaling/latest/userguide/as-scale-based-on-demand.html#as-scaling-adjustment, 2016.

[17] "GoogleCloudPlatform/python-docs-samples," https://github.com/GoogleCloudPlatform/python-docs-samples, 2016.

[18] "Amazon CloudWatch - Cloud & Network Monitoring Services," https://aws.amazon.com/cloudwatch/, 2016.

[19] B. Furht, "Cloud Computing Fundamentals," in *Handbook of Cloud Computing*, B. Furht and A. Escalante, Eds. Springer US, 2010, pp. 3–19.

[20] "Apache JMeter - Apache JMeter™," http://jmeter.apache.org/, 2016.

[21] "JMeter-Plugin," https://jmeter-plugins.org/wiki/UltimateThreadGroup/, 2016.

[22] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, 1st ed. New York, NY, USA: Cambridge University Press, 2013.

[23] B. W. Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica Et Biophysica Acta*, vol. 405, no. 2, pp. 442–451, Oct. 1975.

[24] H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, and S. Leue, "Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples," in *Proceedings of the 2009 Sixth International Conference on the Quantitative Evaluation of Systems*, ser. QEST '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 299–308.

[25] V. Shmatikov, "Probabilistic Analysis of an Anonymity System," *J. Comput. Secur.*, vol. 12, no. 3,4, pp. 355–377, May 2004.

[26] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn, "Probabilistic Model Checking of Complex Biological Pathways," in *Computational Methods in Systems Biology*, ser. Lecture Notes in Computer Science, C. Priami, Ed. Springer Berlin Heidelberg, Oct. 2006, no. 4210, pp. 32–47.

[27] V. A. d. S. Júnior and S. Tahar, "Time Performance Formal Evaluation of Complex Systems," in *Formal Methods: Foundations and Applications*, ser. Lecture Notes in Computer Science, M. Cornélio and B. Roscoe, Eds. Springer International Publishing, Sep. 2015, pp. 162–177.

[28] M. A. S. Netto, C. Cardonha, R. L. F. Cunha, and M. D. Assuncao, "Evaluating Auto-scaling Strategies for Cloud Computing Environments," in *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, Sep. 2014, pp. 187–196.