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

Alexandros Evangelidis[a,*], David Parker[a], Rami Bahsoon[a]

[a]*School of Computer Science, University of Birmingham, United Kingdom*

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 the 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 techniques through a detailed validation process on two public cloud providers, Amazon EC2 and Microsoft Azure, targeting two cloud computing models, Infrastructure as a Service (IaaS) and Platform as a Service (PaaS), respectively. Our experimental results show that the modelling process along with the model itself can be very effective in providing the necessary formal reasoning to cloud application owners with respect to the configuration of their auto-scaling policies, and consequently helping them to specify an auto-scaling policy which could minimise QoS violations.

Keywords: Auto-scaling, Markov models, probabilistic verification

1. 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 [1]. 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.

*Corresponding author

Email addresses: a.evangelidis@cs.bham.ac.uk (Alexandros Evangelidis), d.a.parker@cs.bham.ac.uk (David Parker), r.bahsoon@cs.bham.ac.uk (Rami Bahsoon)

While auto-scaling is an extremely valuable feature for software providers, specifying an *auto-scaling policy* that can guarantee no performance violations will occur is an extremely hard task, and "doomed to fail" [2] unless considerable care is taken. Furthermore, in order for a *rule-based* auto-scaling policy to be properly configured, there has to be an in-depth level of knowledge and a high degree of expertise, which is not necessarily true in practice [3, 1].

Lately, public cloud providers such as Amazon EC2 and Microsoft Azure have increased the flexibility offered to users when defining auto-scaling policies, by allowing combinations of auto-scaling rules for a wide range of metrics. However, this "freedom" of being able to specify multiple auto-scaling rules comes at 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 [4].

In addition, an auto-scaling policy consists not only of performance metrics thresholds, but also of *temporal parameters*, which often seem to be neglected, despite their significance in configuring a good auto-scaling policy. These parameters include the time interval that the auto-scaling mechanism looks back to determine whether to take an auto-scale action, and the duration for which it is prohibited from triggering auto-scale actions after a successful auto-scale request (cool-down period). Since both of these parameters have to be specified by a human operator, it becomes a challenging task to understand the impact of these parameters on performance metrics of the application running on the cloud. It is exactly this impact that we wish to quantitatively analyse.

As noted in [5], 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 formal verification. In particular, we use *probabilistic model checking* [6], which is a formal approach to generating guarantees about quantitative aspects of systems exhibiting probabilistic behaviour, and the tool PRISM [7]. Guarantees are expressed in quantitative extensions of temporal logic and numerical solution of probabilistic models is used to quantify these measures precisely. This approach provides a formal way of quantifying the uncertainty that exists in today's cloud-based systems and 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 Op-*

erating 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.

Our probabilistic model is a discrete-time Markov chain (DTMC), which we specify in the modelling language of the PRISM tool. It 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, considering an Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) scenario running on the Amazon EC2 and Microsoft Azure cloud, respectively. We have made the data used to validate our model publicly available [8].

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 [9]. In a sense, we follow a 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 of a QoS violation. ROC can be used to help the decision maker 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 [10].

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" [9] 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 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 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 [11, 5]. 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 major public cloud providers, Amazon EC2 and Microsoft Azure, and the first usage of ROC analysis to validate such results.

This paper is an extended version of our earlier work on performance modelling and verification of auto-scaling policies [12]. In particular, this paper investigates the generalisability of the approach to a second public cloud provider, Microsoft Azure, dealing with auto-scaling policies at the PaaS, rather than IaaS, level. It also extends the techniques to consider temporal properties of the auto-scaling policies, and how their variation can affect the performance guarantees that can be offered to users.

The remainder of the paper is structured as follows. In Section 2, we provide some background on auto-scaling policies and probabilistic verification. Then, in Section 3, we describe our approach to constructing a probabilistic model of the auto-scaling process. Sections 4 and 5 present the process for applying probabilistic verification to the model and validating the results, respectively. Section 6 discusses the results, and then we conclude by surveying the relevant literature and identifying directions for future work.

2. Preliminaries

2.1. Rule-based auto-scaling policies

An *auto-scaling policy* [4] defines the conditions under which capacity will be added to or removed from a cloud-based system, 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 approaches also being defined as *vertical* (add more RAM or 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 [13].

In a rule-based approach, the application provider has to specify an upper and/or 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* [14]. The performance metrics that public cloud providers usually follow include CPU utilisation, throughput and queue length. 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. As discussed

earlier, apart from the standard auto-scale metrics, there are also other important *temporal* parameters of auto-scaling policies, such as the time interval for looking back and the cool-down period.

2.2. Probabilistic model checking and PRISM

PRISM [7] is a *probabilistic model checker*, which supports the construction and formal quantitative analysis of various 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.

For DTMCs, properties of the model are specified in PRISM using an extension [6] of the temporal logic PCTL (probabilistic computation tree logic) [15]. 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.

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

The states in our model represent the information needed to capture the dynamics of the auto-scaling process and its impact on QoS. Apart from the use of boolean variables which are used to synchronise certain transitions in our model, we have employed clustering methods to summarise the monitored CPU utilisation and response time traces. These clusters are used to characterise the different states of our model, and in the following subsections we provide the details with respect to the clustering procedures followed.

3.1. Clustering

For the model developed for the IaaS use case on Amazon EC2, we standardise the CPU utilisation and response time values by computing their *z-scores* [16]. 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. In a sense, it captures the CPU utilisation and response time variability that exists for a given 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. However, the appropriate value of k is important for having a model that can generalise to other CPU utilisation, and response time traces.

In our model of the IaaS use case on Amazon EC2, after experimenting with different cluster sizes, we have set $k = 5$. For the second type of model which targeted a PaaS use case on Microsoft Azure, a *univariate* clustering method was employed, since we did not take into consideration the response time, and consequently had to deal with one-dimensional data. This allowed us to use the *Ckmeans.1d.dp* algorithm which "guarantees optimality" for data in a 1-D space [17]. The value of k in this case was determined by trying a range of different cluster sizes and picking one with the appropriate *Bayesian Information Criterion (BIC)* (see Figures 1 and 2 for the BIC and the respective clustering plots for the CPU utilisation traces for the auto-scaling policies with the 5 minute cool-down period considered).

3.2. Encoding auto-scaling policies in PRISM

Although two different models have been developed for the Amazon EC2 and Microsoft Azure case, the core principles underlying them are similar. For example, both of the models expect an auto-scaling policy as an input, the creation of their states relies on clustering, and the "philosophy" of how transitions unfold between the states is the same to an extent. In addition, both of the models share the same high level objectives, since they are meant to assist in the auto-scaling decision making process. In the next paragraph, we describe the important building blocks of our models by highlighting their differences and by explaining the reasons of those differences.

For the Amazon EC2 case, 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 specifying the number of instances which will be added; vi) the decrement specifying the number of instances which will be removed; vii) the number of VMs that are currently reserved; viii) the maximum time the model will run; ix) the probability p; and x) the time that it will take for an auto-scale request to be satisfied. 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).

The "free" parameters for the model developed for the Microsoft Azure case are the increment and decrement variables, the number of currently reserved

VMs and the maximum time the model will run. The reason for having fewer constants is to explore the impact on the performance metrics of interest of more "direct" auto-scaling policies, that is policies which do not express the capacity to be added/removed as a percentage of the total capacity, but as an integer which specifies the number of VMs to be added or removed. In addition, if the capacity adjustments are expressed as a percentage, one has to think about the rounding rules that exist, and the conditions that have to be fulfilled when a percentage in capacity has to be added or removed.

In order to be able to develop a representative model for the Amazon EC2 case, there are certain conditions with respect to the addition/removal of VMs that have to match those in realistic clouds. As an example, we show some of those conditions, encoded as formulae in PRISM, and are being invoked in the respective states in our model. For example, our model has to fulfil the following conditions, with respect to the scale-out and scale-in adjustments. Values in the intervals [0..1] and $(-\infty..-1)$ are rounded up, while values in the intervals [-1..0] and $(1..+\infty)$ are rounded down respectively. In Listing 1, we show a sample of the PRISM code for these conditions, expressed as formulae.

Listing 1: Changing capacity formulae

```
formula adjust_c1=(cVMs*s_o_adj_1)>=incr?ceil(cVMs*s_o_adj_1):incr;
formula adjust_f2=(cVMs*s_o_adj_2)>=incr?floor(cVMs*s_o_adj_2):incr;
formula adjust_si_f1=(cVMs*s_in_adj_1)>=decr?ceil(cVMs*s_in_adj_1):decr;
formula adjust_si_f2=(cVMs*s_in_adj_2)>=decr?floor(cVMs*s_in_adj_2):decr;
```

As a result, part of our reasoning when developing the model for the Microsoft Azure use case was to explore how much of this overhead could be avoided, when dealing with auto-scaling policies where the requested capacity is expressed as an integer, denoting the number of VMs to be added/removed.

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. For the Amazon EC2 model, these outcomes, and the respective probability q, are set according to *k-means* clustering which has been run beforehand. In Listing 2 we show a sample of the PRISM code for the transition based on the value of p, and for more details, we refer the interested reader to our previous work [12].

Listing 2: Transition based on probability p

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

For the Microsoft Azure model, there is no probability p associated with certain transitions. This is because we did not want to investigate *what-if* scenarios with respect to the time variability of the auto-scaling policies, rather our focus was on analysing the impact on the QoS of varying the cool-down periods of an auto-scaling policy. Based on this intention, we added specific

guarded commands to certain states in our model in order to prevent "flapping situations" where the auto-scale controller triggers actions continuously [18]. For example, in the states where a scale-in action is to be taken, we evaluate if the resulting CPU utilisation after releasing a VM will be less than the scale-out CPU utilisation threshold, to determine whether the scale-in action will be skipped or not. In Listing 3, we show how this case is expressed in PRISM's guarded commands, and as a result the respective scale-in transition is skipped when the guard is violated.

Listing 3: Guarded commands in scale-in states to avoid "flapping".

```
[scale_in_act1] (...)&( INITIAL_VMS=2 &(cVMs=3 | cVMs=4))&(decr=1)& ( (
    ↪ util* cVMs) / (cVMs-decr) < cpu_threshold )
```

4. Formal verification of auto-scaling policies

In this section we present the verification process for the two cases considered, namely the IaaS case on Amazon EC2 and PaaS case on Microsoft Azure.

4.1. IaaS case on Amazon EC2

The verification process consists of two phases. In the first phase, we generated 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 approximately 26 hours of data collection. 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 1: 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 5.1.1 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 1. 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 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*$= 1$.

4.2. PaaS case on Microsoft Azure

Table 2: Auto-scaling policies for formal verification.

Action	Inc/Dec	Min Util.	Max Util.	VMs	Cool-down
Scale-out	[1]	71%	100%	[1..4]	5 minutes
Scale-in	[1]	0%	39%	[1..4]	5 minutes
Scale-out	[1]	71%	100%	[1..4]	1 minute
Scale-in	[1]	0%	39%	[1..4]	1 minute
Wait	-	40%	70%	[1..4]	-

The verification process for Microsoft Azure follows a similar approach to the one for Amazon EC2, with the difference being that we did not consider response time as part of our QoS metrics, since we focused exclusively on CPU utilisation and how it is affected by the variation of the cool-down period, which is part of an auto-scaling policy. The reason for focusing exclusively on CPU utilisation is the fact that we wanted to restrict our attention to server side metrics, and only to metrics on which the auto-scale decisions are taken. The model was initialised in a similar way to the Amazon EC2 case. In the first phase, we

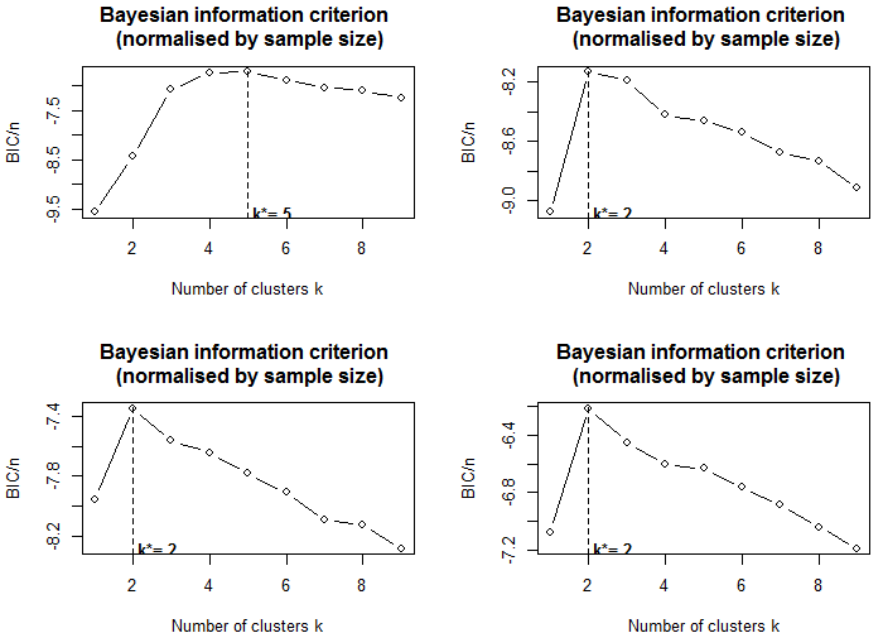


Figure 1: Determining k according to the Bayesian Information Criterion (BIC).

generated load to the VMs and we gathered approximately 400 data points for CPU utilisation for the range of VMs 1 to 4, resulting in approximately 26 hours of CPU utilisation traces that were used to construct our model. For example, in Figure 1 we show the BIC plots that were used to determine the number of clusters per VM number, and in Figure 2 we show the respective (optimal) clustering plots of CPU utilisation after having chosen the number of clusters according to the previous procedure.

Our verification goal is twofold; first we wish to compute the probability that the auto-scaling policies in Table 2 could result in a QoS violation, and then to identify the set of states which have a high probability of transitioning to a "bad" state. By doing that, we show how PRISM can be used to narrow down the search state space to states which are more likely to cause "bad" auto-scaling actions. We define a "bad" auto-scaling decision as an auto-scaling request which, even though it has been successful and the requested capacity has been added/removed, still it has caused a QoS violation. Our main claim here is that properly configuring the temporal parameters is equally important to the proper configuration of the "standard" auto-scaling parameters such as the capacity to be added/removed.

To address the aforementioned concepts, we first identify the set of states where a "bad" auto-scale state is true. This is achieved by adding two respective boolean variables in our model; *bad_scale_in*, and *bad_scale_out* and setting them

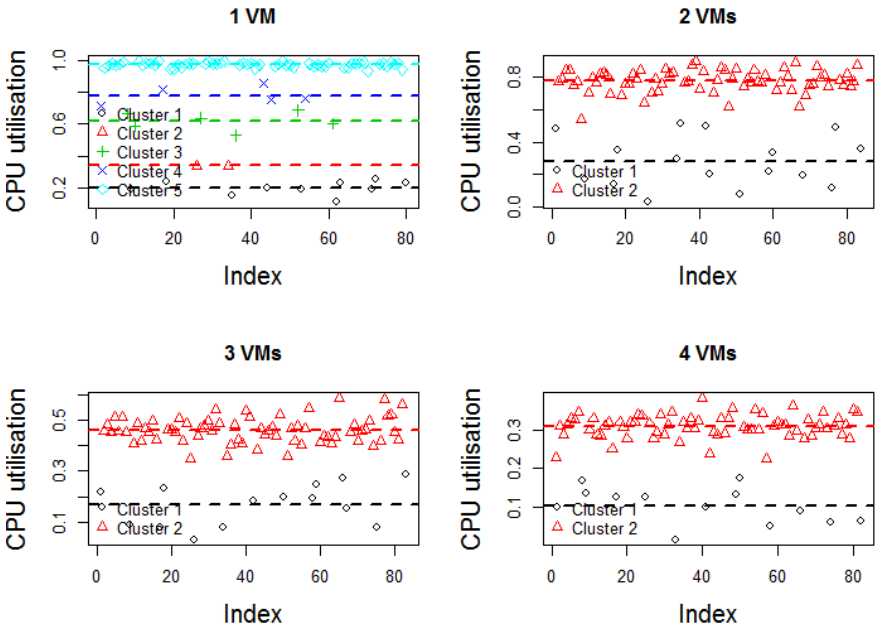


Figure 2: Optimal univariate clustering of CPU utilisation per VM number.

to true and false when certain transitions occur in our model. After this point we can follow two possible verification strategies. We can either use PRISM's *reward* operator R to assign a reward of 1 to states labelled as "bad" and then compute the expected value of "bad" auto-scale decisions over all paths for a given period. An alternative verification strategy is, after having computed the probabilities for the "bad" auto-scale states, to combine PRISM's *filtering* operation, with the temporal operator X, to select the set of states which have a high probability to transition to "bad" auto-scale states. That is, we identify the states that occur one time step *before* a "bad" auto-scale state. For our validation purposes we have adopted the former verification strategy (see Section 5.4). In Listing 5 we show those properties expressed in PRISM.

Listing 5: Properties to be checked

```
R{"bad_auto_scale_state"}=? [F "end"]
P=? [F bad_scale_out=true]
P=? [F bad_scale_in=true]
filter(print, filter(argmax,P=?[X "bad_util_scale_in"]))
filter(print, filter(argmax,P=?[X "bad_util_scale_out"]))
```

5. Model validation

The validation framework consists of three parts. The first is the experimentation setup on the respective public cloud providers, 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 [8].

5.1. Experimentation setup on Amazon EC2

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 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 [19] 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.

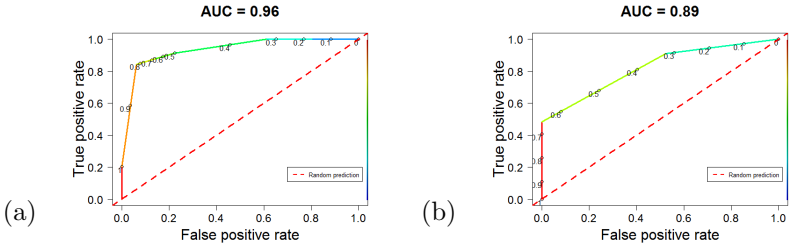
In addition, to monitor and log all the metrics of the auto-scaling group, we have used Amazon EC2's monitoring service, CloudWatch [20]. The performance metrics are averaged over all the VMs. We monitor and gather the performance metric data for each VM number and for each auto-scaling policy. For each of the policies shown in Table 1, we generated load and monitored 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.

5.1.1. 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 [21]. To generate a periodic load we have used Apache JMeter [22], 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* [23] 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.

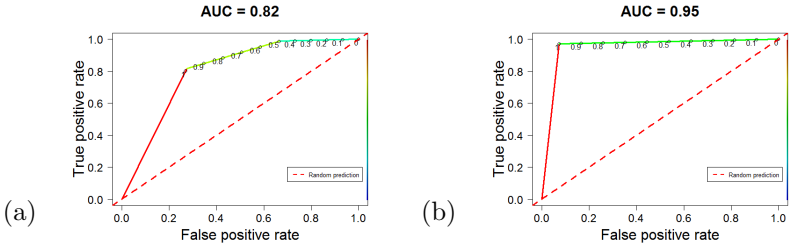
The second type of workload we are considering has a greater degree of randomness, and our aim is to validate our model against an aggressive load pattern, but with an inherent degree of randomness. For this type of workload we make the assumption that HTTP requests arrive according to a Poisson process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



(a) (b)

Figure 3: ROC curves under "periodic" load: (a) CPU util. viol.; (b) resp. time.



(a) (b)

Figure 4: ROC curves under "aggressive" load: (a) CPU util. viol.; (b) resp. time.

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 [24] works as follows: 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.

5.2. 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

Table 3: 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 4: 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

curve and point (0,1). Figures 3–4 show the ROC curves for CPU utilisation and response time under the two load patterns. In Figure 3(a), 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 3–4, 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) [25], which takes values between -1 and +1, indicating a negative and positive correlation between the predicted and the actual value. Tables 3 and 4 show results for this, as well as several other performance measures: overall accuracy of model (ACC); true positive rate (TPR); true negative rate (TNR); false positive rate (FPR); and false negative rate (FNR).

5.3. Experimentation setup on Microsoft Azure

For the experimentation setup on Microsoft Azure, we have deployed the Bakery web application template, developed with ASP.NET and mainly in JavaScript, to the Azure App Service [26]. This allowed us to validate our model under a more realistic and modern web application, while at the same time considering a PaaS cloud computing model (App Service). For this purpose, we have created an App Service Plan, and rented *S1 - Standard* VMs, with 1 core, and 1 GB of RAM. In addition, for our auto-scale plan we considered VMs in the range [1..4] inclusive. The CPU utilisation of the VMs was monitored using Azure's monitoring service on a per-minute basis.

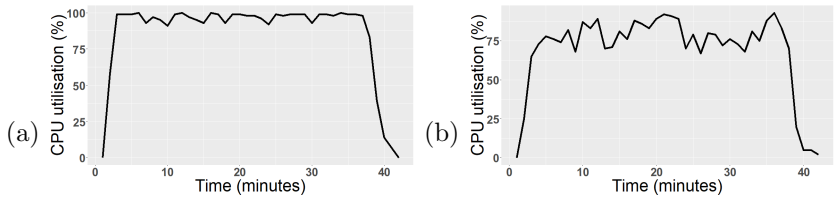


Figure 5: Sample CPU utilisation traces: (a) 1 VM; (b) 2 VMs.

5.3.1. Load profile

In order to generate load we used four Apache JMeter client machines, three virtual machines running Ubuntu Linux 16.04 LTS, and one physical machine running Windows 10. The three VMs were deployed through Microsoft Azure to the Western Europe region and they were of type *Standard DS11 v2*, while the physical machine had 4 cores, and 16 GB of RAM. The test scripts were designed in such a way to stress the web application, by performing a sudden increase to nearly 900 threads, and executing a test scenario for approximately 40 minutes. We have also made the XML test scripts available here [8].

In order to make our test scenario more interesting, we recorded some browsing patterns and injected random delays between consecutive requests. This allowed our test script to be more representative of a realistic case, by considering the *think time* of users when browsing a web page. 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 2, we generate load and monitor our VMs on Microsoft Azure cloud for 40 minutes. Then, we repeat this process 10 times for each auto-scaling policy, resulting in 6.6 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. In Figure 5, we show sample CPU utilisation traces for 1 and 2 VMs.

5.4. Results and model validation

In this section we describe our model validation with respect to what has been discussed in Section 4.2. As a metric to compare the measurements from our model with those from the realistic case, we have used the relative error, which has been used in performance modelling and analysis studies [27]. The relative error is defined as $err = \frac{X_{meas} - X_{sim}}{X_{sim}}$, where the numerator denotes the absolute error. In Table 5, we show the relative error for the expected number of bad auto-scale decisions between our PRISM model, and the measurements taken from our App Service on Microsoft Azure.

Table 5: Relative error for "bad" auto-scale actions.

Auto-scale actions	PRISM	Simulated	Cool-down	Rel. error
Scale-out	1.01	1	5 minutes	1%
Scale-in	0.62	0.6	5 minutes	3.33%
Scale-out	2.287	2.8	1 minute	18%
Scale-in	2.8	2	1 minute	40%

6. Discussion of results

For the verification of auto-scaling policies on Amazon EC2, 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 3–4. 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 3 and 4 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 3), 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 misclassification 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 4), 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 on 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.

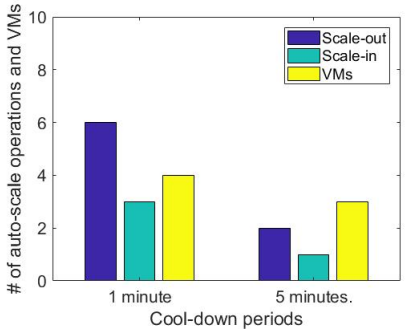


Figure 6: Auto-scale operations and VMs under the different cool-down periods.

For the Microsoft Azure case, our high-level goal was to show the importance of the temporal parameters of an auto-scaling policy (e.g. cool-down period) on tbe performance objectives of an application hosted on the cloud. In addition, we wanted to explore how the non-static parameters of an auto-scaling policy, could be modelled and encoded in the PRISM modelling language. With these goals in mind, we have experimented with the auto-scaling policies shown in Table 2, and in the next paragraph we report on the results shown in Table 5.

For the 5-minute cool-down period in an auto-scaling policy our model captures accurately enough the expected number of bad auto-scale decisions, with the bad scale-out actions being detected more accurately by our model, as the relative error for those actions is 1%. Furthermore, we note that the relative error for the expected number of bad scale-in actions is around 3%. The increase by approximately 2% in the relative error can be explained by the fact that the CPU utilisation traces which were used to validate our model, had a slightly greater variability in the scale-in actions for a certain number of VMs.

Also, an important observation that we made by examining our sample CPU utilisation traces is that in most of the cases there existed a pattern with respect to how auto-scale actions are being triggered given the workload used as an input. In addition, from a modelling perspective the stabilisation period of five minutes after an auto-scale action has been triggered, made the correspondence, regarding the duration of each state, between the sampled CPU utilisation traces and our model easier. The negative effect of a decreasing stabilisation period on the accuracy of our model can be seen by the increased relative error for the bad scale-out and scale-in actions. This is because by choosing the minimum cool-down period of one minute we are allowing for the CPU utilisation to stabilise between the consecutive auto-scale requests.

The additional complexity in capturing the temporal pattern of an auto-scale controller which is programmed to fire auto-scale requests consecutively stems from the fact that it becomes a formidable task to detect the actual sending time of an auto-scale request, and the respective time by which the auto-scale request was satisfied. The reasons above provide an explanation regarding the increased relative error under the 1-minute cool-down period. However, we are partially satisfied since our model managed to detect the bad scale-in actions in

a moderately effective manner.

As part of our exploratory analysis, we show in Figure 6 the negative effect of having cool-down periods of short time intervals. For instance, the average number of scale-out operations for the 1-minute cool-down period is 6 compared to 2 for the 1-minute cool-down period. More importantly, the extra scale-out operations did not provide any significant benefit in minimising the QoS violations. In addition, the scale-in actions were 3 under the 1-minute cool-down compared to 1 for the 5-minute cool-down period. Finally, we note that the maximum number of VMs which seemed to be required under the 1-minute cool-down was 4 compared to 3 under the 5-minute cool-down.

7. Related work

Probabilistic model checking has been employed with great success in recent years to verify and analyse properties of systems that manifest uncertainty. Domains include automotive systems, security, biology and many others. Lately, there has been an increased interest by researchers in applying probabilistic model checking to 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 clear 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 [11] used probabilistic model checking to verify the performance of live migrations in the IaaS layer [11]. They assume that migration requests are distributed in a uniform way, which is not necessarily true in practice [28].

In addition, [5] 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 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 [29], 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. In addition, interesting work applying formal methods to prevent QoS violations, but not specifically to auto-scaling, can be found in [30]. The other differentiating factor in our work is that we perform an extensive validation of our models on two major public cloud providers (Amazon EC2, Microsoft Azure). 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 try to

infer the different outcomes that could happen. In general, however, we note that the use of probabilistic model checking for pragmatic cloud use cases is still at its infancy; our research aims towards bridging this gap.

8. Conclusion and future work

We presented a novel probabilistic verification scheme, followed by an extensive and robust validation on VMs rented on two major public cloud providers using two different cloud computing models, IaaS and PaaS for Amazon EC2, and Microsoft Azure respectively. This is, to the best of our knowledge, the first in-depth study of auto-scaling policies, which is conducted on public cloud providers, 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, for the Amazon EC2 case, 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.

For the Microsoft Azure case we have focused on the temporal parameters of an auto-scaling policy, by considering cool-down periods of one and five minutes. Then, our goal was to show the usefulness of the PRISM model checker to compute the expected number of bad auto-scale decisions, and we presented our results having taken into account CPU utilisation as a performance metric. Moreover, we showed the difficulties that manifest in the modelling process as the cool-down period becomes shorter, and highlighted the difficult problem of attributing a (meaningful) duration to the states in the model in order to match to an extent the realistic situation.

In this work, we dealt mainly with the dynamics of an auto-scaling policy by varying the increments and the initial VMs in operation, cool-down periods as part of the controllable parameters. Also, we have made a first step towards quantifying the impact of the temporal properties of auto-scaling policies on QoS metrics, such as CPU utilisation. To this end, we have made the correspondence between those temporal patterns to a probabilistic model, and shown how probabilistic model checking can play a significant and more active role in dealing with pragmatic cloud computing problems. As future work, we will experiment with cool-down periods of various durations, and conduct further research in the context of auto-scaling policies which dynamically change.

Acknowledgements. The first author is partially supported by an EPSRC-funded Ph.D. studentship (award ref: 1576386). This work has been partially

supported by a Microsoft Azure Sponsorship (offer subscription id: f1d14833-9e4b-40bb-bf42-a0bf3b2baf8b). The second author is part-funded by the PRINCESS project (contract FA8750-16-C-0045), under the DARPA BRASS programme.

References

[1] M. Wajahat, A. Gandhi, A. Karve, A. Kochut, Using machine learning for black-box autoscaling, in: Proc. IGSC'16, 2016, pp. 1–8.
[2] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, I. Truck, From Data Center Resource Allocation to Control Theory and Back, in: Proc. CLOUD'11, 2010, pp. 410–417.
[3] R. Han, L. Guo, M. M. Ghanem, Y. Guo, Lightweight Resource Scaling for Cloud Applications, in: Proc. CCGrid'12, 2012, pp. 644–651.
[4] H. Ghanbari, B. Simmons, M. Litoiu, G. Iszlai, Exploring Alternative Approaches to Implement an Elasticity Policy, in: Proc. CLOUD'11, 2011, pp. 716–723.
[5] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, S. Sioutas, Dependable Horizontal Scaling Based on Probabilistic Model Checking, in: Proc. CCGrid'15, 2015, pp. 31–40.
[6] M. Kwiatkowska, G. Norman, D. Parker, Stochastic Model Checking, in: Proc. SFM'07, no. 4486, Springer Berlin Heidelberg, 2007, pp. 220–270.
[7] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: Proc. CAV'11, Vol. 6806 of LNCS, Springer, 2011, pp. 585–591.
[8] Suporting material, `http://www.prismmodelchecker.org/files/fgcs-autoscaling/`.
[9] T. Fawcett, An Introduction to ROC Analysis, Pattern Recogn. Lett. 27 (8).
[10] W. J. Krzanowski, D. J. Hand, ROC Curves for Continuous Data, 1st Edition, Chapman & Hall/CRC, 2009.
[11] S. Kikuchi, Y. Matsumoto, Performance Modeling of Concurrent Live Migration Operations in Cloud Computing Systems Using PRISM Probabilistic Model Checker, in: Proc. CLOUD'11, 2011, pp. 49–56.
[12] A. Evangelidis, D. Parker, R. Bahsoon, Performance modelling and verification of cloud-based auto-scaling policies, in: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017, pp. 355–364. `doi:10.1109/CCGRID.2017.39`.
[13] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, G. Iszlai, Optimal Autoscaling in a IaaS Cloud, in: Proc. ICAC'12, ACM, New York, NY, USA, 2012, pp. 173–178.
[14] L. M. Vaquero, L. Rodero-Merino, R. Buyya, Dynamically Scaling Applications in the Cloud, SIGCOMM Comput. Commun. Rev. 41 (1) (2011) 45–52.
[15] H. Hansson, B. Jonsson, A logic for reasoning about time and reliability, Formal Aspects of Computing 6 (5) (1994) 512–535.
[16] C. Romesburg, Cluster Analysis for Researchers, Lulu Press, 2004.
[17] H. Wang, M. Song, Ckmeans.1d.dp: Optimal k-means Clustering in One Dimension by Dynamic Programming, The R journal 3 (2) (2011) 29–33.
[18] Best practices for autoscale - Microsoft Azure, `https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/insights-autoscale-best-practices/` (2017).
[19] `https://github.com/GoogleCloudPlatform/python-docs-samples`.
[20] Amazon CloudWatch, `https://aws.amazon.com/cloudwatch/` (2016).
[21] B. Furht, Cloud Computing Fundamentals, in: B. Furht, A. Escalante (Eds.), Handbook of Cloud Computing, Springer US, 2010, pp. 3–19.
[22] Apache JMeter - Apache JMeter™, `http://jmeter.apache.org/` (2016).
[23] JMeter-Plugin, `https://jmeter-plugins.org/wiki/UltimateThreadGroup/` (2016).
[24] M. Harchol-Balter, Performance Modeling and Design of Computer Systems: Queueing Theory in Action, 1st Edition, Cambridge University Press, New York, NY, USA, 2013.
[25] B. W. Matthews, Comparison of the predicted and observed secondary structure of T4 phage lysozyme, Biochimica Et Biophysica Acta 405 (2) (1975) 442–451.
[26] Bakery template - Bakery Template™, `https://azuremarketplace.microsoft.com/en-us/marketplace/apps/Microsoft.Bakery?tab=Overview/` (2017).

[27] N. Huber, S. Becker, C. Rathfelder, J. Schweflinghaus, R. H. Reussner, Performance modeling in industry: A case study on storage virtualization, in: Proc. ICSE'10, 2010.
[28] V. A. d. S. Júnior, S. Tahar, Time Performance Formal Evaluation of Complex Systems, in: M. Cornélio, B. Roscoe (Eds.), Proc. SBMF'15, Springer, 2015, pp. 162–177.
[29] M. A. S. Netto, C. Cardonha, R. L. F. Cunha, M. D. Assuncao, Evaluating Auto-scaling Strategies for Cloud Computing Environments, in: Procc. MASCOTS'14, 2014.
[30] F. Raimondi, J. Skene, W. Emmerich, Efficient online monitoring of web-service slas, in: Proc. SIGSOFT'08/FSE-16, ACM, 2008, pp. 170–180.