Modeling and Verifying Microservice Autoscaling Using Probabilistic Model Checking

Siti Nuraishah Agos Jawaddi
Universiti Teknologi MARA (UiTM), Shah Alam
Azlan Ismail (azlanismail@uitm.edu.my)
Universiti Teknologi MARA (UiTM), Shah Alam
Valeria Cardellini
University of Rome Tor Vergata

Research Article

Keywords: Autoscaling, Energy consumption, Markov Decision Process, Microservices, Probabilistic Model Checking, Scalability

Posted Date: May 25th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1682990/v1

RESEARCH

Modeling and Verifying Microservice Autoscaling Using Probabilistic Model Checking

Siti Nuraishah Agos Jawaddi[2], Azlan Ismail[1,2*] and Valeria Cardellini[3]

[*]Correspondence:
azlanismail@uitm.edu.my

[1]Institute for Big Data Analytics and Artificial Intelligence (IBDAAI), Kompleks Al-Khawarizmi, Universiti Teknologi MARA (UiTM), 40450, Shah Alam, Selangor, Malaysia

Full list of author information is available at the end of the article

Abstract

Microservices can independently adjust their capacity to match demand while the autoscaling feature in cloud computing facilitates the users (i.e., developers) to provision resources required by their applications with less human intervention. Kubernetes is one of the well-known technologies used to deploy microservice-based applications and many autoscaling methods have been proposed to improve the behavior of its Horizontal Pod Autoscaler (HPA). Despite many research efforts have been recently devoted to investigate microservice autoscaling, there is still a lack of studies that consider the correctness of the scaling decision as well as the effect of the scaling process on host energy consumption and system scalability factors. Therefore, in this work we aim to take into account formal verification in the microservice autoscaling decision-making process by utilizing Markov Decision Process (MDP) and probabilistic model checking. To this end, we propose five MDP model variations, inspired by the scaling behavior of Kubernetes-based HPA, analyze the performance of the models from a combination of metrics. The *Base Model* is built by considering the CPU utilization metric in decision making, while the other models extend it by including several additional metrics to enhance the decision (i.e., latency, response time, energy consumption, and performance change). We use the PRISM-games model checker for the analysis purpose by verifying the properties specified in Probabilistic Computation Tree Logic (PCTL). Through our experiments, the decision made by the *Full Model* which considers all the metrics has outperformed the other models in terms of minimizing the energy consumption and leading to a good scalability level (i.e. scalability near to 1).

Keywords: Autoscaling; Energy consumption; Markov Decision Process; Microservices; Probabilistic Model Checking; Scalability

1 Introduction

The design, development, and operation of microservice-based applications keeps increasing for cloud-native services due to their autonomous capabilities. The microservices architectural style improves the traditional monolith one in terms of many aspects, including scalability, reliability, availability as well as dynamic development, and can be deployed either at the infrastructure or the application level [1, 2]. Unfortunately, it also contributes to the increment of cloud workloads. However, the autoscaling feature allows a microservice-based system to automatically control its resource provisioning according to specific demands when facing

varying loads. Such microservice autoscaling can be performed according to three dimensions, namely horizontal scaling, which allows replication of the microservice instances, vertical scaling, which scales the amount of computing resources, and the combination of both approaches.

Many autoscaling solutions have been proposed to address highly dynamic and complex cloud workloads, as surveyed in [3]. Apart from the more common reactive autoscaling policies, which are often relatively simple and rely on user-defined scaling rules (e.g., CPU utilization thresholds), proactive autoscaling strategies aim to anticipate the decision so to minimize the time taken to enact autoscaling. However, the process of determining the best autoscaling decision-making policy may be time-consuming, besides involving costs to rent cloud resources to perform testing on the microservice system. Meanwhile, Bushong et al. [4] have reviewed several methods that can be used to analyze microservices, being model-based analysis with model checking one of them. Model-based analysis introduces an abstraction approach to capture certain system concerns, whilst model checking is useful in finding errors at design stage to prevent any occurrence of them during the deployment stage.

Currently, several studies have exploited model checking for model-based analysis to address certain concerns in microservices, either with non-probabilistic or probabilistic model checking. For example, the works in [5, 6, 7] applied non-probabilistic model checking that concerns the microservice interaction, resiliency and security, respectively. Meanwhile, other works [8, 9, 10] implemented probabilistic model checking to investigate microservice resiliency patterns, orchestration, and autoscaling behaviors, respectively. On the other hand, Ray and Banerjee [11] specifically chose Markov Decision Process (MDP) to model the fault-recovery procedure, while Kochovski et al. [12] used MDP to model the process of selecting deployment options for microservice-based applications. However, to the best of our knowledge the studies in literature that analyze the microservice autoscaling decision-making are hardly found for the MDP-driven probabilistic model checking.

In this article, we propose a method to analyze the microservice autoscaling decision-making process using probabilistic model checking, which consists of three main phases, namely, modeling, encoding and experimentation. First, we model the autoscaling decision-making process based on the scaling behavior of Kubernetes' Horizontal Pod Autoscaler (HPA) using Markov Decision Process (MDP), as it can capture the probabilistic and non-deterministic aspects of the modeled problem. Second, we encode five variations of MDP models using the PRISM-games model checker [13], which is a tool that allows modeling and analysis of several probabilistic models including MDPs. Besides, the PRISM-games model checker also allows computation of optimal probabilistic and expected values for verification, which fulfills our needs to formally verify the autoscaling policies that meet the optimal strategy. Third, we verify the MDP models on properties specified in Probabilistic Computation Tree Logic (PCTL) to determine the optimal scaling decisions. We have selected PCTL due to its capability of quantifying properties in probability form.

In summary, the main contributions of this article are as follows:

- a method to analyze microservice autoscaling decision making using probabilistic model checking by means of the MDP specification;
- five variation of MDP models that consider several metrics in the decision making of Kubernetes' horizontal autoscaling process (i.e., CPU utilization, latency, response time, energy consumption, and performance change);
- the analysis of performance results of each MDP model based on scalability and energy consumption.

The rest of this paper is organized as follows. In Section 2 we provide background on cloud-based microservice autoscaling and Markov Decision Process(MDP). In Section 3 we present the proposed autoscaling decision-making model. In Section 4 we discuss the experimentation result. In Section 5 we summarize the related works and highlight their key findings. Finally, we conclude and present directions for future work in Section 6.

2 Background

2.1 Microservice Autoscaling

Microservice is an architectural approach to implement a distributed application as small and loosely coupled services with the capabilities to independently scale itself [1]. This section discusses the fundamental knowledge of microservice autoscaling in terms of the scaling dimension, scaling strategy, architectural reference model based on MAPE-K loop, as well as the involved deployment tools.

2.1.1 Scaling Dimension and Strategy

Microservice autoscaling is built based on scaling dimensions (i.e., horizontal and vertical) and scheduling strategy (i.e., reactive and proactive). Horizontal scaling is the dominant approach used to scale cloud-native microservice applications by adding or removing replicas or instances of the same service on demand. Meanwhile, during vertical scaling, an appropriate amount of computing resources is assigned to each replica by increasing or decreasing the CPU shares or the memory. For example, Kwan et al. [14] combined horizontal and vertical scaling to adapt the microservice deployment.

On the other hand, the reactive approach adapts to the current demands, while a proactive approach predicts the future demands and plans in advance for appropriate resource provisioning. The works in [15, 16] are examples of reactive and proactive autoscaling techniques, respectively. Liu et al. [15] proposed a reactive autoscaling policy based on fuzzy logic to dynamically adjust the scaling threshold and cluster size, while Choi et al. [16] relied on a graph neural network to proactively scale the microservice chain.

2.1.2 MAPE-K Autoscaling

A self-adaptive software system adjusts its behavior according to the presence of certain operating conditions and MAPE-K (*Monitor, Analyze, Plan and Execute with Knowledge*) is a well known architectural pattern to organize the adaptation loop [17]. A framework to drive the autoscaling of microservice applications can

thus be designed according to this loop, e.g., [18]. In this work, we focus on the Plan component, where several properties are used to verify the decision-making process through formal verification to guarantee the correctness of the autoscaling policies.

2.1.3 Kubernetes' Horizontal Pod Autoscaler

Each microservice is deployed at the container level to address the portability challenge faced by distributed applications [19]. To simplify the deployment, management, and execution of containerized applications, orchestration tools are adopted. Kubernetes is the most popular open-source container orchestration platform, developed by Google for automated pod deployment and cluster resource management. *Cluster* refers to a set of node machines for running the containerized applications, while *pod* is the smallest deployment unit in Kubernetes. It consists of one or more tightly coupled containers that are co-located and scaled as an atomic entity.

The scalability of containerized applications can be controlled at the pod granularity, both horizontally and vertically, and at the node granularity. The Horizontal Pod Autoscaler (HPA) scales the number of pods within the specified nodes [20], the Vertical Pod Autoscaler oversees the resource allocation of existing pods, and the Cluster Autoscaler adjusts the number of nodes in the specified cluster [21]. In this work, we focus on the HPA implementation in Kubernetes.

HPA helps to improve cloud resource usage by promoting pods replication. Many research works regarding HPA and Kubernetes have been recently proposed, e.g., [22, 18, 23]. Nguyen et al. [22] provided some insights about HPA, Balla et al [23] proposed an adaptive two-dimensional scaling policy for Kubernetes pods, and Rossi et al. [18] presented a self-adaptive hierarchical architecture and related autoscaling policies for microservice-based applications and used HPA as a baseline policy for performance comparison.

HPA follows a MAPE-K control loop implemented by the `kube-controller-manager`, which at each specified period gathers pod resource metrics and used them to assist the scaling decisions regarding a single microservice deployment [20]. On each Kubernetes worker node, the resource metrics are collected by the `kubelet` node agent and the `kube-controller-manager` gets these metrics through the Kubernetes API server. The number of pods is required to be within the range of minimum and maximum pods allowed in a system. The autoscaling decision is determined according to a classic threshold-based approach, which scales the number of current pods according to the ratio between the observed value and the target value of the considered metric. Specifically, the desired number of replicas is computed using Eq. 1:

$$desiredReplicas = \lceil currentReplicas * \frac{currentMetricValue}{desiredMetricValue} \rceil \quad (1)$$

where $currentMetricValue$ and $desiredMetricValue$ refer to the monitored value of the considered metric (e.g., CPU utilization) and to its target value, respectively. The pod CPU utilization is computed as the ratio between the total actual CPU

usage of all containers in a pod and the CPU resource requested by the pod. Apart from CPU and memory utilization, HPA also supports custom metrics collected from external monitoring tools (e.g., Prometheus) such as HTTP request rate.

2.2 Microservice Autoscaling Challenges

In this section, we elaborate two key challenges of microservice autoscaling for carrying out this study.

2.2.1 Scalability

From a cloud computing perspective, a system is considered scalable if it is able to increase its capacity by adding the quantity of its consumed lower layer services [24]. Meanwhile, horizontal scaling is the commonly used approach to improve the scalability of a microservice-based system in which the pods or containers that are already allocated with a certain amount of resources (e.g., CPU and memory) are scaled out or in to ensure the system maintains a good performance while processing the varying and unpredictable workloads.

Apart from the benefit, several challenges need to be faced when scaling the microservice-based system due to its modular characteristics. Firstly, the heterogeneous microservices can cause the microservice autoscaling process to be costly due to different implementation of resources [25]. Secondly, cascading effect needs to be avoided to prevent the degrades in performance of the subsequent microservices in the same chain [16]. Thirdly, CPU and memory utilization metrics are not enough in ensuring the microservice-based application is evenly scaled in a non-CPU or memory-intensive-based application [26]. Hence, we can see there is a need to build a formal framework for microservice autoscaling to tackle these challenges. The formal framework should enable the developers to consider various metrics in scaling microservices at a different level and without worrying about the high cost of heterogeneous implementation of microservices.

The work by Khazaei et al. [27] is an example that scales the microservice as a whole application by concerning both micros as well as macro resources (i.e., containers and VMs) perspectives. Khaleq and Ra [26] implemented horizontal autoscaling at the pod level based on Kubernetes and considered response time as QoS metric instead of CPU utilization alone to measure the effectiveness of the autoscaling policy. Meanwhile, Zhong and Buyya [25] addressed the scaling of microservices on heterogeneous resources by including the task allocation strategy. However, none of these works formally verified the correctness of the autoscaling policy.

2.2.2 Energy Efficiency

The increase in deployment rate of data-driven applications has led to the opening of large data centers which causes data centers to be one of the fastest-growing sources that demand electrical energy [28]. Figure 1 illustrates the scenario of energy consumed by the data center through two types of server hosts, namely, active and idle hosts. The active hosts are deployed with VMs that perform tasks and utilize the computing resources, whilst the VMs in idle hosts do not perform task processing. However, electrical energy still needs to be consumed to activate these

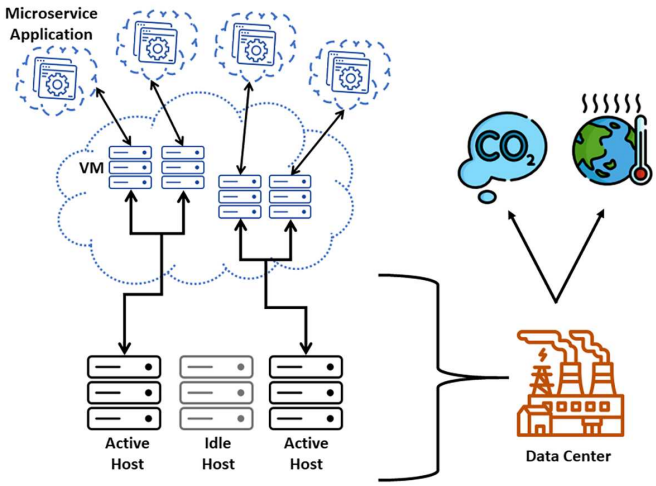


Figure 1: Energy Consumption Scenario in Data Center

hosts. Meanwhile, power consumption is included in the total cost of the data center, while a huge amount of the energy consumption also contributes to the release of carbon footprints [29]. Therefore, the need to reduce energy consumption levels has become an important concern in sustaining the data center. However, there is another shortcoming of this approach, in which the accuracy of the used autoscaling policy has not yet been verified.

Furthermore, resource management approaches can be applied to promote energy saving in data centers [30], while CPU utilization is identified as one of the important elements that contribute to the changes in energy consumption of the hosts [31]. Due to that, CPU metric is often considered in resource scaling techniques, such as those implemented in Kubernetes-based HPA in which the formula in Equation 1 is used to compute the number of desired pod replicas. Examples of studies that address the energy efficiency issue in microservices are presented in [32] and [33], which both conclude that microservice implementation helps in reducing the energy consumption through its efficient utilization of computing resources. However, there is still another shortcoming of this approach, in which the correctness of the scaling algorithm used has not been verified.

2.3 Probabilistic Model Checking

This section discusses the fundamental implementation of MDP in probabilistic model checking as well as the properties use to verify the decision-making process of the MDP model.

2.3.1 Markov Decision Process (MDP)

MDP is a mathematical framework that supports the modeling of decision-making in many situations. It can also be considered as the extension of the Discrete-Time Markov Chain (DTMC) due to the additional non-determinism provided by the model apart from probabilistic behavior [34]. This characteristic fits our model

requirement as it needs to decide on multiple scaling options based on certain conditions to reach optimum goals for the energy and scalability factors. The MDP is represented as a tuple of $M = \{S, s_{init}, Act, P, AP, L\}$ where:

- S is a set of states;
- s_{init} is an initial state;
- Act is a set of action labels;
- $P : S \times Act \rightarrow Dist(S)$ is the partial transition probability function, where $Dist(S)$ refers to the set of all discrete probability distributions over S;
- AP is a set of atomic propositions;
- $L \times S \rightarrow 2^{|AP|}$ is a labelling function that assigns to each state $s \in S$, a set $L(s)$ of atomic propositions.

2.3.2 Properties

There are two types of standard properties used to verify decisions made by the MDP model, namely the probabilistic and expected reachability properties [35]. The first property focuses on determining the probability of reaching a certain state that satisfied the predicate needed for quantitative analysis of the model. The latter is needed to perform a cost-based analysis of the model. Both properties are implemented according to the decision strategy which either minimizes or maximizes certain attributes of the targeted process.

Furthermore, there are two ways of reasoning about the properties of MDP models using PRISM logic [36], namely, probabilistic computation tree logic (PCTL) and linear temporal logic (LTL) state properties, where LTL is easier in expressing properties for non-probabilistic case compared to PCTL [35]. Meanwhile, the syntax of these PRISM logics is derived from four main elements, which are the properties (i.e., probability (P), reward (R)), the LTL formula, the reward objective, and the single instance of P or R operators (e.g., true) [36]. Equation 2 shows an example of the expected reward property which questions the maximum time taken for the step to reach the final state.

$$R\{"time"\}max =? [Fstep = final] \tag{2}$$

3 MDP-Based Analysis Method

This section discusses the analysis process flow as well as the details of microservice autoscaling decision-making models.

3.1 Overview

Figure 2 illustrates the flow of the method to analyze the autoscaling decision-making process using probabilistic model checking, which comprises three main phases, namely, modeling, encoding, and experimentation. The analysis starts with the modeling phase, which consists of the process of identifying required information from the autoscaling decision-making scenario and translating them into modeling

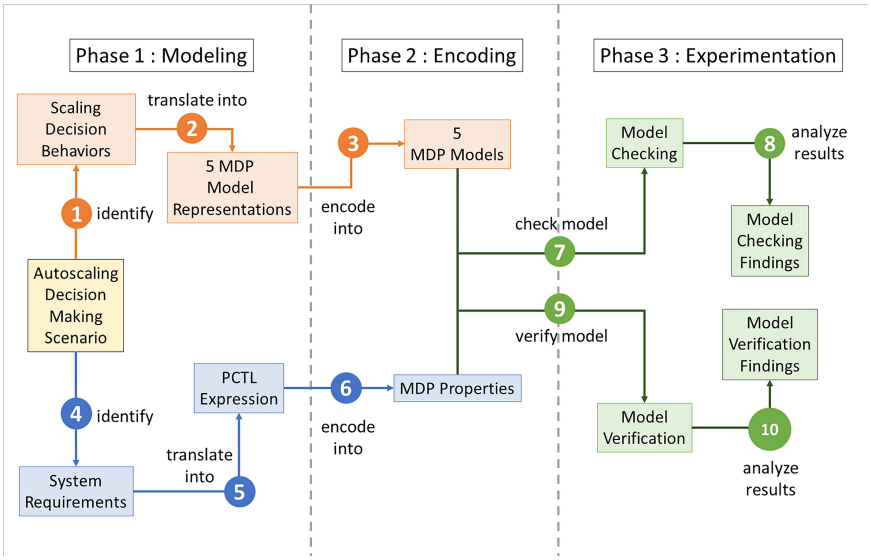


Figure 2: Method of Analyzing Autoscaling Decision Making Process with MDP-driven Probabilistic Model Checking

representation. It is followed by the encoding phase, where the model representations are encoded according to the targeted specification; in this paper, we focus on MDP. Finally, during the experimentation phase, the results for model checking and verification are generated and analyzed.

The flow of the process starts by determining the scaling behavior of Kubernetes' HPA through its algorithm and translating it into the MDP model representation (step 1). Then, the model representation is encoded with MDP specification using PRISM language (step 2). After that, the process continues with identifying the trade-offs factors to be analyzed on the MDP model (step 3) and translating the information into Probabilistic Computational Tree Logic (PCTL) expression (step 4), before encoding them into MDP properties using the PRISM language (step 5). After completing the encoding process, the MDP models are checked to determine whether it is functioning as required (step 6), before analyzing the results and performing any amendments if needed (step 7). When the MDP models are able to function properly, the models are then verified against the properties (step 8). Lastly, the verification results are analyzed to conclude some findings (step 9).

3.2 Phase 1: Modeling

In this phase, the required information is gathered based on an autoscaling decision-making scenario. Firstly, we identified the scaling decision behaviors, which are related to the metrics that should be considered in the decision making, scaling condition, and the number of pods to add or remove from a targeted microservice. By using the information gathered, we translated the scaling decision behaviors into 5 MDP model representations with consideration of different metrics. Secondly, we identify the system requirements and translate them into Probabilistic Computation Tree Logic (PCTL) expression.

Table 1 lists the extracted information from the scenario that is used to manually translate the scaling decision process into MDP model representation, as illustrated in Figure 3. The concepts and descriptions refer to the relevant key concepts used to model the process and the description of the concepts, respectively. The modeling roles are the role of each concept in the process that is mapped to the model, while modeling elements are the symbol used to interpret the concepts. From the scenario, we categorized the concepts into three roles, namely, variable, constant, and action. The variable is the element that will be updated during the decision-making process, while the constant is treated as a fixed threshold value along the process. Meanwhile, the action refers to the scaling event. Based on Figure 3, the Base Model (as explained in Section 3.2.1) considers CPU utilization in its decision-making process, while the extended models (as explained in Section 3.2.2) takes into account other additional metrics apart from utilization.

Table 1: Extracted Information from Autoscaling Decision Making Scenario

Concepts	**Descriptions**	**Modeling Roles**	**Modeling Elements**
Number of pods	Number of pods in the scenario	Variable	S
Maximum pods	Maximum number of pods allowed for the system	Constant	$max\ pod$
Minimum pods	Minimum number of pods allowed for the system	Constant	$min\ pod$
Latency	Delay between request arrival and system response	Variable	lat
Response time	Time to process the request and respond	Variable	rt
Maximum response time	Maximum time to process the request and respond	Variable	$maxRt$
Power consumption	Energy consumed by host on which microservice is deployed	Variable	pow
Maximum power	Maximum power consumption of the active host	Constant	$maxPower$
Scalability	Value of the system performance change microservice	Variable	$scalability$
Scale-out	Add pods to current number of pods	Action	$scale\_out$
Scale-in	Remove pods from current number of pods	Action	$scale\_in$
Do nothing	Current number of pods is unchanged	Action	do_not

In addition, the behaviors of the autoscaling decision-making process have also been extracted and listed in Table 2 while the system requirements are listed in Table 3. The behaviors and requirements are labeled with a unique id (e.g., DB1

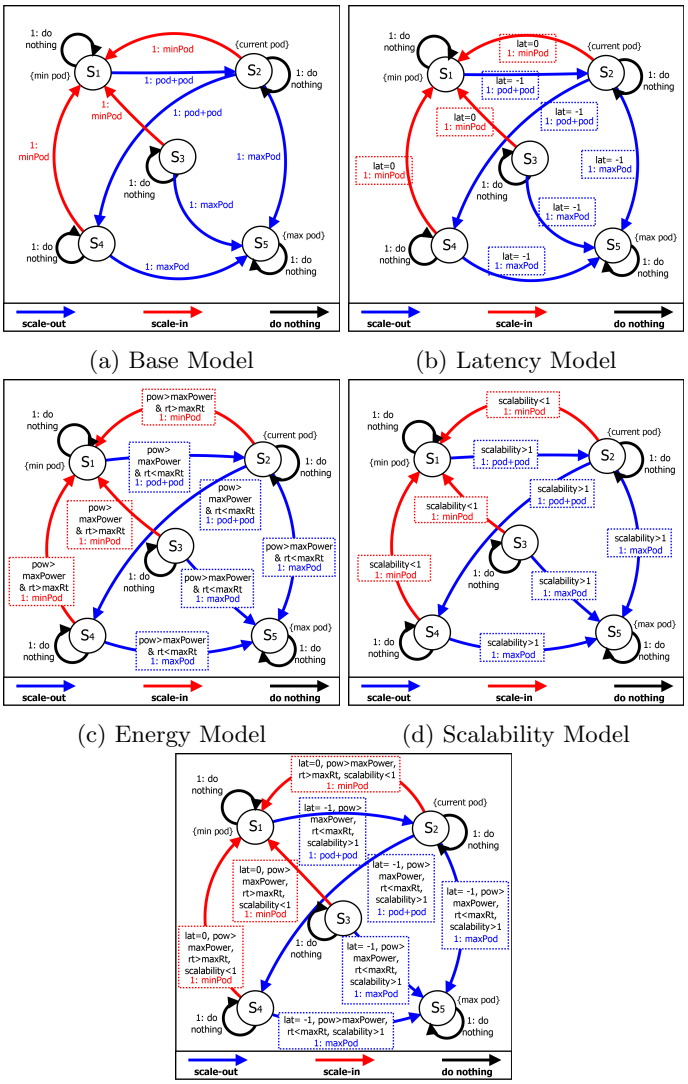
Table 2: Extracted Scaling Decision Behaviors from Scenario

Behavior ID	**Description of Behavior**
DB1	Number of pods should meet the number of desired replicas
DB2	Number of pods should be less than or equal to maximum pods allowed for the system
DB3	100% of currently running replicas is removed during scale-in process
DB4	100% of currently running replicas is added during scale-out process
DB5	40 pods is added during scale-out process

Table 3: Extracted System Requirements from Scenario

Requirement ID	**Description of Requirement**
SR1	Scalability lower than 1 when number of pods equal to desired replica
SR2	Scalability greater than 1 when number of pods equal to desired replica
SR3	Host energy consumption when number of pods equal to desired replica or less than the maximum pods
SR4	Host energy consumption less than maximum energy of the host

and SR1), where DBx refers to the unique ID of scaling decision behavior, while SRx is the ID for system requirements. The behavior information is used to construct the probabilistic model, whilst the system requirements are used to construct properties to be verified on the MDP models. DB1 and DB2 are the conditions to be fulfilled during the scaling process, in which the number of pods allocated is adjusted to meet the number of desired replicas while at the same time is still below its upper bound. DB3 until DB5 hold the information related to the number of resources that are added to or removed from the current number of pod replicas when either the scale-out or scale-in process is triggered. Meanwhile, SR1 and SR2 hold the information regarding a situation in which the system is assumed to be in the low scalability state and high scalability state, respectively. Requirement SR3 is related to the level of energy consumed by the host when the pod allocation of the system is either equal to desired replicas or less than the maximum pods. Lastly, SR4 is the



(a) Base Model (b) Latency Model

(c) Energy Model (d) Scalability Model

(e) Full Model

Figure 3: MDP Model Illustrations

requirement to observe whether the decision made by the models causes the host to consume less power than the maximum power expected for the host.

3.2.1 Base Model

Figure 3(a) shows the autoscaling process of the proposed MDP models, which is adopted and adapted from the MDP model proposed by Naskos et al.[37] and default scaling behavior by HPA discussed in [20]. The running example for the *Base Model* involves three state labels, namely, *current pod*, *min pod*, and *max pod*. Meanwhile, the scaling actions are represented by the colored arrows, in which the blue arrow refers to scale-out (i.e., *add pods*), the red arrow refers to scale-in (i.e., *remove pods*), and the black arrow refers to when the no scaling action is performed and thus the number of pods is left unchanged.

Apart from that, the scaling decision is influenced by the CPU utilization range and the scaling is terminated if the system met either one condition of these three in which if the current utilization range is between 40% to 60%, the maximum number of pods or desired replicas is reached. The utilization is separated into three ranges that lead to different scaling actions, in which the range from 0% to 40% triggers scale-in, 40% to 60% triggers no scaling action, and 60% to 100% triggers scale-out. When performing scale-out, the current number of pods is doubled or added with another 4 pods; however, if the desired number of replicas exceeds the maximum allowed number, it will be set to the maximum number of pods instead. On the other hand, when performing scale-in action, the current number of pods is assigned to the minimum number allowed for the system.

For instance, the current state S_2 in Figure 3(a), which is allocated with 2 pods, can either scale in or out depending on the utilization condition. In this case, if the desired replica determined for S_2 is greater than the maximum number of pods allowed for the system, the number of pods is scaled out to the maximum pods (i.e., S_5) whilst the current number of pods is added with another 4 pods if the counted desired replica is less than 40 pods. However, if the system utilization is currently between 0% to 40%, the number of pods is scaled in to the minimum number of pods allowed for the system (i.e., S_1). Then, if the system removed its pod down to the minimum level (i.e., S_1), the system can be scaled up using either one scale-out step explained above.

3.2.2 Model Variations

In this section we present the extended versions of the *Base Model*, which have been desgined following an incremental approach. In all the extended models, the CPU utilization remains the fundamental metric considered to perform the scaling decision process. They differ from their own characteristics, where the decision made by the *Latency Model (LM)* is influenced by the latency as additional factor, the *Energy Model (EM)* takes also into account the energy consumption, the *Scalability Model (SM)* considers the scalability value, and finally the *Full Model (FM)* exploits all the factors (i.e., latency, energy consumption, and scalability value).

The *Latency Model* enhances the *Base Model* by considering the system latency and response time in its decision-making. Figure 3(b) illustrates the running example for the *Latency Model*. The idea of involving latency and response time in the scaling process is adapted from Cámara et al. [38], who formerly studied the adaptation that considers latency information. In this paper, the latency can either be assigned with -1, which refers to the inactive status of the pods, 0, which refers to a system with no latency, as well as values greater than 0, which represent the latency of the system. Meanwhile, the value for the initial response time, maximum response time, and updated response time is gained through the user input as the value is treated as if it is received from the Analyzer component. On the other hand, the running response time started with the initial response time value and is only replaced with the updated response time value if there is any scaling action that has taken place. The idea from [18] is adopted and adapted where the scale-out process is performed when the microservice response time is greater than the threshold response time

value and the response time will reduce after the scaling process. Meanwhile, if the microservice response time is less than the threshold value, a scale-in process is performed to reduce resource wastage.

In addition, based on the example in Figure 3(b), the scaling actions are also performed if the current response time is greater than the maximum response time allowed for the system, while the latency value is determined to let the system add more pods or remove pods from the system.

The second model variation, that is the *Energy Model*, makes use of the encoded *Base Model* and *Latency Model*, which means it considers utilization, latency, and response time as its metrics. Figure 3(c) shows the running example for the *Energy Model*, which inherits the same process as the *Latency Model*. In this model, the energy consumption of the running host is taken into account along with the latency and response time before performing the scale-out process. The increase in the number of pods contributes to reduce their utilization level, which eventually influences the total energy consumed by the host.

The *Scalability Model* differs from *Energy Model* as it takes into account the current scalability value of the system. Figure 3(d) depicts its MDP, in which the scalability is observed so to enable the system to either add or reduce the number of pods, where scale-out is triggered if the scalability value is greater than 1 and vice versa if the scalability is less than 1. Finally, in contrast with the three model variations, the *Full Model* shown in Figure 3(e) combines all the elements mentioned previously (i.e., CPU utilization, latency, response time, energy consumption, and scalability) in determining the appropriate scaling decision for the tested system.

3.2.3 Model Rewards

The models are analyzed with similar quality attributes, namely, *Expected Energy Consumption* and *Expected Scalability Value*.

The *Expected Energy Consumption* refers to the cumulative total energy consumed by the host during each system state. The formula in Equation 3 is adapted from [31] and used to computed the reward values for the specified states. The reward value is computed if the number of pods assigned to the system is either equal to the number of desired replicas or less than its maximum pods.

$$Energy = Idle\ Power + (Utilization(\%) * Maximum\ Power) \tag{3}$$

On the other hand, the *Expected Scalability Value* refers to the performance change of the system under two conditions (i.e., high scalability and low scalability). The scalability value is computed using the formulas in Equations 4 and 5, which are both adapted from [39].

$$\begin{aligned} PRR_1 &= \frac{1}{WaitingTime\ \times\ Current\ Number\ of\ Pods} \\ PRR_2 &= \frac{1}{MaximumTime\ \times\ Maximum\ Number\ of\ Pods} \end{aligned} \tag{4}$$

$$PC = \frac{PRR_1 \times Demand}{PRR_2 \times Maximum\ Demand} \tag{5}$$

PRR_1 and PRR_2 refer to the performance/resource ratio rate in the current state and future state, respectively, while PC stands for the performance change of the two states. In this work, the elements of the formulas are interpreted with several variables. The variables selected to compute the current performance/resource ratio in this model are the time step (t), the current number of pods (s) and demand (d), while for the future ratio the variables considered are the maximum time ($maxTime$), the maximum number of pods ($maxPod$) and maximum demand ($maxDemand$). The future ratios are assigned with these variables due to their fixed values. The changes between the two ratios are then computed to observe the scalability of the system. A truly scalable system should have a performance change value that is close to 1, while the system with good scalability will reach the value 1 [39]. Therefore, we grouped the observations on the system scalability into two perspectives, namely, high scalability and low scalability. High scalability refers to the condition where the system scalability is over the good scalability level and vice versa for the low scalability. In this work, we restrict the values for high scalability of the system in the range 1 to 5, while the low scalability value is restricted within 0 to 1. The two scalability rewards are computed if the current number of pods met the counted desired replica.

In the MDP models, there are four reward structures as labeled in Table 4 encoded to determine the scalability value of the system and energy consumed by the host for the specified states. Listing 1 shows the general structure of the implemented reward, in which the command is guarded with the combination of state conditions. The rewards for scalability and energy consumption are assigned with values computed using the formulas in Equations 5 and 3, respectively.

```
guard : reward;
```

Listing 1: Reward Structure

Table 4: Reward Label

Reward Label	**Description**
low_scalability	System state cause scalability less than 1 after scaling
high_scalability	System state cause scalability greater than 1 after scaling
energy_consumption	Host power consumption at current state

3.3 Phase 2: Encoding

The input for the second phase is the MDP model representations of the *Base Model* and the model variations illustrated in Figure 3 as well as the labeled rewards in Table 4.

3.3.1 Encoding MDP Model

The decision model is composed of two modules (i.e., concurrent processes) that act as the *kubelet* and *autoscaler* components involved in the HPA decision-making process. The parameters listed in Table 5 are received through user input, while the parameters in Table 6 are assigned with fixed values due to several reasons. The maximum time step, as well as the maximum demand, are set for computing the system scalability in which the two values are assumed as the waiting time and demands for the performance resource ratio of the future state, as discussed in Section 3.2.3. The maximum response time acts as the threshold that controls the scale-in and scale-out process in certain commands, while the updated response time is a value that replaces the previous response time after a certain scaling action is performed. Followed by the value for a maximum power of an active host as well as the idle host, which are adopted from [31] and target utilization, which represents the target resource utilization for the microservice system that is used to calculate the desired replicas. Lastly, a fixed value of CPU capacity requested by the pods is used to update the system utilization, after adding or removing pods based on the formula in Equation 6.

$$updated\ CPUutilization\ = \lceil \frac{current\ CPU\ utilization}{CPU\ requested\ by\ pod} \rceil \tag{6}$$

Table 5: Input Parameter Constants

Constants	**Description**
init_pod	Initial number of pods
init_rt	Initial response time
maxPod	Maximum number of pods allowed for the system

Table 6: Fixed Parameter Constants

Constants	**Description**
TAU	Period duration
maxTime	Maximum time steps
maxDemand	Maximum demands that can be received by the system
minPod	Minimum number of pods allowed for the system
maxRt	Maximum response time
up_rt	Updated response time
idlePower	Host power at idle state
maxPower	Host maximum power at active state
cpu_request	CPU resource request by pods
target_util	Target resource utilization
u_util	Updated utilization after scaling process

The *kubelet* module is assumed to hold the metrics information collected from the services within the worker node (i.e., CPU utilization, latency, demand, power consumption) and the number of pods to be assigned to the registered services. The process starts by determining the range number of pods that are currently assigned to the service, in which the range group is separated into two groups. The first group refers to the number of pods within the range of 0 to 50 pods, while the second group holds the metrics information if the number of pods is within 50 to the limit number of pods. Furthermore, the idea of a probabilistic tree has been adopted from Moreno et al. [40] and adapted to randomly select the metrics information of the services once the group is determined. The Extended Pearson-Tukey (EP-T) three-point approximation [41] used in this model consists of three points that refer to the percentiles of the estimation distribution (i.e., 5th, 50th, 95th percentiles) with probabilities of 0.185, 0.63, and 0.185, respectively. We assume that the values of the metrics are received from the Monitoring component integrated into kubelet (i.e., cAdvisor). Apart from selecting the monitored data, this module also receives the updated number of pods from the *autoscaler*, in which the kubelet module will ensure the targeted service is occupied according to the updated number of pods.

The *autoscaler* is the core module of the decision model, as it decides when to scale the number of pods and how many pods should be assigned to the targeted services by making use of the metrics information held by the kubelet module. Meanwhile, in terms of scaling behavior, there are two options available for the scale-out process and one option for scale-in as mentioned in [20] as the default behavior of HPA. If the module is triggered to add more pods, it can either double the pods or add 4 pods to the current number of pods. For instance, if the counted desired replica is less than the maximum number of pods allowed for the system, the current number of pods held by the autoscaler module will be doubled, otherwise, the current number of the pod is assigned to the value of the maximum pods. Then, if the desired replica count is less than 40, the current number of pod is added with 4 pods for that time step, otherwise the current number of pod will be doubled. On the other hand, if the number of pods is greater than the maximum number of pods, the scale-in process is triggered and the current number of pods is reduced to the value of the minimum pods allowed for the system, whilst the scaling is terminated when the number of pods is either equal to the value of the maximum pods or the desired replicas.

By making use of the model representation in Figure 3, the module is encoded as shown in Listing 2 which consists of local variables that hold metrics information of the services after performing the scaling decision (i.e., updated number of pods, utilization, time steps, response time and latency). Followed by commands that enable the scaling decision made based on the selected metrics to be examined. The *Base Model* decision is set up on the range of CPU thresholds, while *Latency Model* is influenced by an additional metric (i.e., latency) apart from the CPU utilization. Then, *Energy Model* considers the combination of the two metrics with energy consumption for its scaling decision, while *Scalability Model* replaces the energy consumption with scalability. Lastly, *Full Model* performs its scaling decision based on all the metrics mentioned. Listing 3 to Listing 5 show the additional codes implemented into the *Base Model* in Listing 2 for the purpose mentioned above.

```
module kubelet
demand : [0..maxDemand] init 0;     pod : [0..limitPod] init init_pod;
l : [−1..maxLat];      u : [0..100] init 0;
pow : [0..1000000] init 0;

[do_not] true → (pod′=current_pod) & (u′=util) & (pow′=ceil(power)) ;

[] (pod>0 & pod<=50) →
 0.185 : (demand′=demand_a) & (l′=lat_a) & (u′=util_a) & (pow′=pow_a) +
 0.630 : (demand′=demand_b) & (l′=lat_b) & (u′=util_b) & (pow′=pow_b) +
 0.185 : (demand′=demand_c) & (l′=lat_c) & (u′=util_c) & (pow′=pow_c) ;

[] (pod>50 & pod<=limitPod) →
 0.185 : (demand′=demand_d) & (l′=lat_d) & (u′=util_d) & (pow′=pow_d) +
 0.630 : (demand′=demand_e) & (l′=lat_e) & (u′=util_e) & (pow′=pow_e) +
 0.185 : (demand′=demand_f) & (l′=lat_f) & (u′=util_f) & (pow′=pow_f) ;
endmodule

module autoscaler
current_pod : [minPod..maxPod] init minPod;
util : [0..100] init 0;
t : [0..maxTime] init 0;
rt : [0..maxTime] init init_rt;
lat : [−1..maxLat] init 0;
rt : [0..maxTime] init init_rt;

[scale_out] (60>=u & u<=100) & (pod<desired_replica)
 & (t + TAU < maxTime) →
 (current_pod′=desired_replica<maxPod?pod + pod : maxPod)
 & (t′=t+TAU) & (util′=u>util?u_util : util);

[scale_out] (60>=u & u<=100) & (pod<desired_replica)
 & (t + TAU < maxTime) →
 (current_pod′=desired_replica<40?pod + 40 : pod + pod)
 & (t′=t+TAU) & (util′=u>util?u_util : util);

[do_not] (40>=u & u<=60) | (current_pod=maxPod) |
 (current_pod=desired_replica) → (current_pod′=current_pod);

[scale_in] (0>=u & u<=60) & (pod>maxPod) & (t + TAU < maxTime)
 → (current_pod′=desired_replica<minPod?minPod : minPod)
 & (t′=t+TAU) & (util′=u>util?u_util : util);
endmodule
```

Listing 2: Module

```
[] (pod<=maxPod) & (l!=0) → (lat′=l>0?l − 1 : lat);

[scale_in] (pod>0) & (lat=0) & (rt<maxRt) & (t + TAU < maxTime) →
 (current_pod′=desired_replica<minPod?minPod : minPod)
 & (t′=t+TAU) & (util′=u>util?u_util : util) & (lat= − 1);

[scale_out] (pod>0) & (lat= − 1) & (rt>maxRt) & (t + TAU < maxTime) →
 (current_pod′=desired_replica<maxPod?pod + pod : maxPod)
 & (t′=t+TAU) & (util′=u>util?u_util : util) & (lat′=l>0?l − 1 : lat)
 (rt′=up_rt);

[scale_out] (pod>0) & (lat= − 1) & (rt>maxRt) & (t + TAU < maxTime) →
 (current_pod′=desired_replica<40?pod + 40 : pod + pod)
 & (t′=t+TAU) & (util′=u>util?u_util : util) & (lat′=l>0?l − 1 : lat);
 (rt′=up_rt);
```

Listing 3: Additional Commands for Latency Model

```
[scale_out] (pow>maxPower) & (rt>maxRt) & (t + TAU < maxTime) →
 (current_pod′=desired_replica<maxPod?pod + pod : maxPod)
 & (t′=t+TAU) & (util′=u>util?u_util : util) & (lat′=l>0?l − 1 : lat)
 (rt′=up_rt);

[scale_in] (pow>maxPower) & (rt<maxRt) & (t + TAU < maxTime) →
 (current_pod′=desired_replica<minPod?minPod : minPod)
 & (t′=t+TAU) & (util′=u>util?u_util : util) & (lat= − 1) & (rt′=up_rt);
```

Listing 4: Additional Commands for Energy Model

$[scale\_out]\ (scalability{>}1)\ \&\ (rt{>}maxRt \mid lat{=}-1)\ \&\ (l{>}0)\ \&$
$(t + TAU < maxTime)\ \rightarrow$
$(current\_pod'{=}desired\_replica{<}maxPod?pod + pod : maxPod)\ \&$
$(t'{=}t{+}TAU)\ \&\ (util'{=}u{>}util?u\_util : util)\ \&\ (lat'{=}l{>}0?l - 1 : lat)\ \&$
$(rt'{=}up\_rt);$

$[scale\_in]\ (scalability{<}1)\ \&\ (rt{<}maxRt \mid lat{=}0)\ \&\ (t + TAU < maxTime)$
$\rightarrow\ (current\_pod'{=}desired\_replica{<}minPod?minPod : minPod)$
$\&\ (t'{=}t{+}TAU)\ \&\ (util'{=}u{>}util?u\_util : util)\ \&\ (lat{=}-1)$
$(rt'{=}up\_rt);$

Listing 5: Additional Commands for Scalability Model

3.3.2 Encoding Properties

There are two types of properties used to verify the autoscaling decision made by the MDP models, namely the probabilistic properties and expected reward properties. The decision is verified from the maximum and minimum perspectives of expected energy consumption, expected scalability frequency, and probability of the decision leading to either high or low energy consumption.

The information gained in Section 3.2.3 is referred and several matters have been questioned to formulate the properties in Table 7, there are *"What is the scalability value reached by the system when implementing the scaling decisions?"*, *"How much is energy consumed by the host when implementing the scaling decisions?"*, and *"Which model can lower the host energy consumption?"*. The reward properties are used to determine the expected maximum and minimum cumulative value of energy consumed by the host within 1000 time units as well as the scalability values of the system under two states (i.e., high, low). Meanwhile, the probabilistic properties result in the probability of the decision made by each model causing the energy consumption level to be lower than the maximum power of the host.

Table 7: Properties

Req. ID	Type	Properties Specification	Description
SR1	Reward	R{"less_scalability"}max=? [C $\leq$ 1000]	Maximum cumulative for scalability value ≤ 1 and > 0 in 1000 time units
SR2	Reward	R{"high_scalability"}max=? [C $\leq$ 1000]	Maximum cumulative for scalability value > 1 and < 5 in 1000 time units
SR3	Reward	R{"energy_consumption"}max=? [C $\leq$ 1000]	Maximum cumulative for energy consumption in 1000 time units
	Reward	R{"energy_consumption"}min=? [C $\leq$ 1000]	Minimum cumulative for energy consumption in 1000 time units
SR4	Probabilistic	Pmax=? [F $\leq$ 1000 power $<$ maxPower]	Maximum probability that host power consumption is less than its maximum power in 1000 time units
	Probabilistic	Pmin=? [F $\leq$ 1000 power $<$ maxPower]	Minimum probability that host power consumption is less than its maximum power in 1000 time units

3.4 Phase 3: Experimentation

The main objective of our experiments is to analyze the impact of a decision made by the MDP models on the host energy consumption and the scalability of the microservice system. The analysis is meant to clarify the following lists of research questions:

- *RQ1*: Does the involvement of metrics latency, response time, energy consumption and scalability in the scaling algorithm impact the energy consumed by the host and the scalability of the system?
- *RQ2*: How do demands received by the system affect the scalability of the system?
- *RQ3*: Does an increase in the number of resources affect the energy consumption of the hosts according to the *Base Model* and the model variations?

3.4.1 Experimental Setup

Two resource allocation scenarios are used to perform this experiment, namely, low resource allocation and high resource allocation to handle the unpredictable resource demands. Then, each scenario is further divided into two situations in which the SUT is either initially allocated with a small number of pods or a large number of pods. Table 8 shows the parameter values that are varied to represent the scenarios while the values for other parameters are kept constant in all configuration settings. The configuration values are input by the user into the PRISM-games model checker in which Setting 1 and 2 represent the situation with less resource allocation as the maximum number of pods is restricted less than the one in Setting 3 and Setting 4. Meanwhile, the initial response time for a configuration with more resource allocation is assumed to be shorter than the configuration with fewer resources allocated. Table 9 listed several parameters that are kept constant in this experiment for all the MDP models while the range of resource demands for the MDP models is restricted between 100 to 1000 to match the resource allocation.

Table 8: Experiment Configuration

Setting	**Initial Pod**	**Maximum Pod**	**Initial Response Time (s)**
1	10	100	7
2	50	100	6
3	10	200	5
4	50	200	4

Table 9: Constant Parameter

Category	**Parameter**	**Value**
Demand	Maximum number of demand	1000
Pod	Minimum number of pod	1
	Limit number of pod	1000
	CPU requested by a pod	3%
Time	Period duration	2s
	Maximum response time	5s
	Updated response time	2s
Power	Maximum power of active host	249W
	Power of idle host	170W

3.4.2 Model Checking and Verification

The MDP models as well as the properties are compulsory to perform the model checking and verification process. During model checking, the values stated in Table 8 are input into the models as early information about the targeted microservice. The models are then built to ensure no deadlock is present. However, if any deadlock is present, a specific property provided by the PRISM-games model checker (i.e., *filter(print, "deadlock")*) is used to identify the states that cause the deadlock. Based on the received information, necessary actions are taken to fix the issue and if the models do not show any deadlock once built, they can be further analyzed in the verification process. The results for model checking consist of the number of states, transitions, and choices available in each model, the reachability time as well as the time to construct the model.

After successfully passing the model checking process, the MDP model is verified using the properties mentioned in Table 2.3.2. Properties that hold the requirements of SR1 until SR3 provide results in the form of expected value for system scalability and energy consumed by the host while properties that represent SR4 provide results in probability form between the range 0 to 1. The results for both processes are further discussed in Section 4.

4 Results and Discussion

In this section, we discuss the experimentation results of the MDP models and analyze the data from two perspectives, namely, energy consumption and scalability.

The model checking results are shown in Table 10, which have been categorized into the number of states, transitions, and choices available in the model, as well as the time taken to verify the properties. Meanwhile, the expected energy consumption, high scalability value, and low scalability value refer to the verification results. The energy consumption is determined by using two optimal strategies (i.e., maximum and minimum), while the maximum value for high and low scalability of each configuration is observed in the experiment. The scalability value is obtained by dividing the cumulative scalability value by 1000, due to 1000 time units.

Based on the observation made in Table 10, as expected the *Base Model* produces the least number of states, transitions, and choices for the decision-making process as well as the verification time, while the *Full Model* yields the highest values for those four elements. This is due to the restrictions made by the different models, where the *Base Model* only considers CPU utilization metrics to perform scaling decisions while the *Full Model* considers all the metrics examined in this paper (i.e., utilization, latency, response time, energy consumption, performance change).

The verification results are analyzed further to answer the research questions proposed earlier and requirements SR1 and SR2 are discussed in Section 4.1, whilst SR3 in Section 4.2. Meanwhile, properties for SR4 give value 1 as output, which means that all the MDP models have a full probability to cause the host energy consumption level to be lower than the maximum host power.

Table 10: Model Checking Results [Note: M - Model, LM - Latency Model, EM - Energy Model, SM - Scalability Model, FM - Full Model, S - Setting, St - State, Ts - Transition, Ch - Choice, TVT - Total Verification Time, MxEC - Max. Energy Consumed, MnEC - Min. Energy Consumed, HSV - High Scalability Value, LSV - Low Scalability Value]

M	S	St	Ts	Ch	TVT (s)	MxEC (W)	MnEC (W)	HSV	LSV
Base	1	226597	890464	437270	145.891	376781.03	170000.00	2.80	0.35
	2	226457	889940	437026	154.035	379782.77	170000.00	2.73	0.34
	3	221677	868324	424970	139.015	376781.03	170000.00	2.56	0
	4	221467	867485	424551	160.559	379272.32	170000.00	2.48	0
LM	1	977500	4765970	2810970	725.915	401444.56	170000.00	4.45	0.56
	2	977039	4763899	2809821	707.841	405247.46	170000.00	4.32	0.54
	3	948964	4609022	2711094	700.182	401444.56	170000.00	4.06	0
	4	2270878	11021238	6479482	1369.573	404597.44	170000.00	3.94	0
EM	1	4182173	22794850	14430504	2695.168	402337.58	11532.54	4.47	0.57
	2	4185923	22815840	14443994	3091.297	405721.06	2254.25	4.37	0.55
	3	948964	4609022	2711094	670.07	401444.56	170000.00	4.06	0
	4	3925325	21404386	13553736	2432.617	404597.44	170000.00	3.94	0
SM	1	4064390	21360387	13231607	2686.105	402647.50	11065.39	4.47	0.57
	2	4063707	21357247	13229833	2188.942	405735.74	2254.25	4.38	0.55
	3	3347837	16458507	9762833	1689.582	401444.56	64796.26	4.06	0
	4	5250249	26310758	15810260	2872.959	405117.98	33905.53	3.99	0
FM	1	6096050	34552914	22360814	4013.969	403459.17	6202.97	4.47	0.57
	2	6107025	34613057	22399007	3323.761	405735.74	1184.75	4.44	0.56
	3	5052207	27579572	17475158	2949.311	401444.56	57052.18	4.06	0
	4	7622234	42948912	27704444	4256.112	405117.98	3708.60	4.05	0

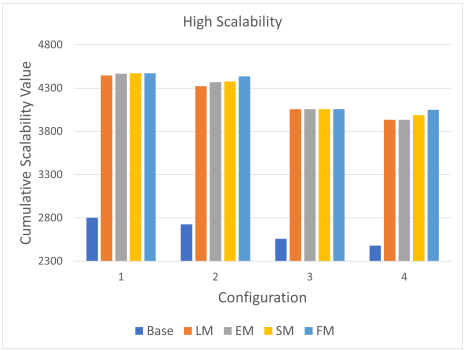
4.1 Scalability Analysis

The scalability of the system is determined by verifying the third and fourth properties listed in Table 2.3.2 on the MDP models followed by analyzing the values with *RQ1* and *RQ2*. The result shown in Figure 4 answered both questions.

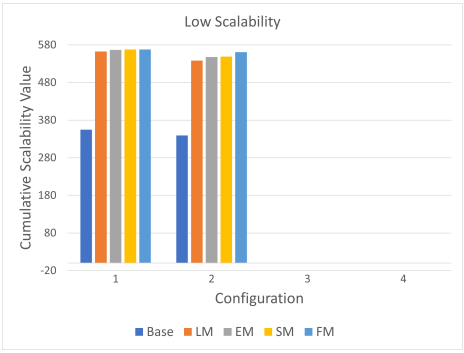
The *RQ1* is answered based on the observation in Figure 4(a) in which Configuration 4 has the least cumulative value for high scalability followed by Configuration 3, 2, and 1. Besides showing that the scalability value is lower compared to others, it also means that the system is less likely to be in the high scalability state compared with other configurations. Meanwhile, in terms of model perspective, the *Base Model* yields the least value which may be due to the less frequent scale-out process performed by the model. On the other hand, the decision made by the extended models leads the system to a high scalability state due to numerous scale-out actions performed.

The result displayed by Figures 4(a) and 4(b) have been analyzed to answer *RQ2*. All configurations manage to have a scalability value greater than 1 (i.e., high scalability) which means that the pods allocation is greater than resource demand or the other way round. Meanwhile, only the scalability of Configuration 1 and 2 are able to get nearer to value 1 in which value 1 refers to the good scalability level. The two observed situations may be caused by the restricted range of resource demands that were kept constant for all the models during the experimental setup and Configuration 1 as well Configuration 2 are mutual in terms of their maximum number of pods allowed for the system. This shows that the number of pods assigned for the two settings almost met the resource demands. In addition, the *Full Model*

is better in leading the system to reach the scalability near the value 1, followed by *Scalability Model*, *Energy Model*, and *Latency Model*.



(a) Maximum High Scalability



(b) Maximum Low Scalability

Figure 4: Expected Cumulative Scalability Value Across Models

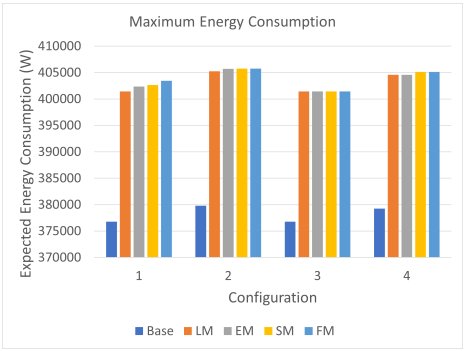
4.2 Energy Consumption Analysis

The host energy consumption results from the decision made by the MDP models are analyzed by verifying the first and second reward properties in Table 7 and the results displayed by Figure 5 answered the *RQ1* and *RQ3*.

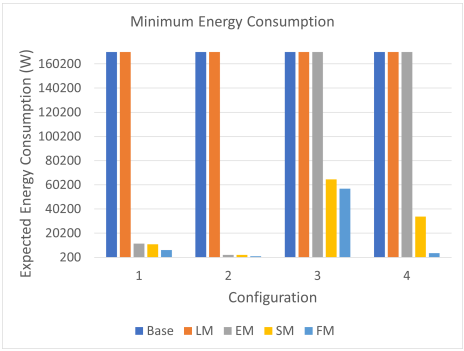
The number of resources affects the energy consumed by the host in different manners in which the consumption level is higher when more pods are deployed to the system as more resource is utilized to fulfill the demands. As can be seen in Figure 5(a), Configuration 2 and 4 causes the host to consume more energy as both settings started by allocating a larger number of pods compared to Configuration 1 and 3. Meanwhile, in the perspective of MDP models, the *Base Model* is able to minimize the energy consumption by controlling the pods' allocation based on the CPU utilization, however, decisions made by *Latency Model* until *Full Model* contribute to higher energy consumption. This may occur due to the *Base Model* only considers one metric to make scaling decisions while other models are influenced by several additional metrics. Therefore, the frequency of the extended models to add or remove pods from the system is higher compared to the *Base Model*.

In contrast, Figure 5(b) shows that *Full Model*, which considers all examined factors (i.e., CPU utilization, latency, response time, energy, scalability), is able to mini-

mize the energy consumption to the lowest level compared to other models. This is because the system needs to fulfill more conditions in order to be allocated with more or fewer pods. Finally, among the five models, the *Base Model* and *Full Model* achieve the most distinct energy consumption level, where the *Full Model* is able to maximize and minimize the energy consumption of the host compared to the *Base Model*.



(a) Maximum Energy Consumption



(b) Minimum Energy Consumption

Figure 5: Expected Cumulative Energy Consumption Across Model

5 Related Work

A systematic review has been done by Bushong et al. [4], who classify the recently published approaches and techniques to analyze microservice systems. Several methods have been identified and categorized as static, dynamic, the combination of static and dynamic, model-based, graph-based, and pattern-based analysis. Meanwhile, in model-based analysis, the abstract models are constructed to represent part of the system concern in a simplified form [4]. In addition, the model checking technique is one of the methods used in this analysis and is known for its capability to verify the qualitative properties of many systems [35]. Hence, in this section, we only highlight the related works that addressed the implementation of non-probabilistic model checking, probabilistic model checking, and MDP-driven probabilistic model checking.

5.1 Non-Probabilistic Model Checking Approach

Non-probabilistic model checking implies the assume-guarantee paradigm [34] and several studies have implemented the approach to analyze certain aspects related to microservices. Dai et al. [5] analyze the interaction behavior of the complex microservice system by translating the property interaction soundness in Linear Temporal Logic (LTL) formula make use of the Process Analysis Toolkit (PAT) tool for verification. Meanwhile, McZara et al. [6] use a design structure matrix to model the dependencies between microservices and analyze the matrix to determine the dependencies that should be considered to build a high resiliency microservice system. Furthermore, Chondamrongkul et al. [7] perform security analysis on a microservice architecture model, in which the structure is built in Ontology Web Language (OWL) and the behavior is built-in Architecture Description Language (ADL). The structure model is then reasoned by the Ontology Reasoner to generate properties to prove the security scenarios followed by verifying the properties of the behavioral model using the PAT model checker.

5.2 Probabilistic Model Checking Approach

Probabilistic model checking aims to provide correctness guarantees for probabilistic system models and several studies that implement the technique with microservices have been identified. A study by Mendonça et al. [8] analyze the resiliency patterns of microservice systems by utilizing Continuous-time Markov Chain (CTMC) to model various resiliency patterns. The behavior of the critical concepts is captured and analyzed with two quality attributes that are calculated using the CTMC reward structure in the PRISM tool. Another work is in [42] which also implements the same foundation as [8] (i.e., using CTMC for modeling in the PRISM tool) introduces a modeling framework for the reliability of microservice-based applications with a service mesh that is separated into micro-model and macro-model perspectives. Meanwhile, the work by Su et al. [9] focuses on analyzing the performance of microservice orchestration systems by modeling and analyzing the performance monitoring framework using Parametric DTMC and PRISM model checker respectively. Burroughs [10] focuses on modeling the microservice scaling behavior in Kubernetes Cluster at runtime. The model is then analyzed by capturing two aspects of behaviors, namely, failure points of pod replicas within a given deployment and possible deployment configuration of pods if any resource competition is determined respectively. Lastly, the verification process is performed by using a generator tool called Waikato Analysis Tool for Events in Reactive Systems (WATERS). Despite all these works, none of them considers the combination of energy consumption and scalability factors in their analysis using probabilistic model checking.

5.3 MDP-driven Probabilistic Model Checking Approach

MDP-driven model checking has been also applied to analyze microservice in cloud computing, e.g., [11, 12]. The study in [11] has made use of MDP to address fault-recovery issues for microservice-based applications in which a fault-recovery procedure is modeled to determine which Multi-Access Edge Computing (MEC) servers should be selected to re-deploy microservice application when a failure occurs. On the other hand, Kochovski et al. [12] implement MDP to model the process of se-

lecting a deployment option for the cloud-based microservice system where the best option is determined based on its ranking. However, to the best of our knowledge, there is no work tackled the autoscaling for microservice aspect.

Our work also utilizes MDP to model microservice autoscaling decision making process and specifically focus on analyzing the system performance based on energy and scalability perspectives. Even though our model is the simplified representation of the decision-making process, which is based on the typical Horizontal Pod Autoscaler, we believe that verifying the autoscaling decision of the microservice system at the abstraction level by using a model-based approach will help to reduce the time and the costs of renting the actual cloud resources for the purpose of automatically analyzing the decision models.

6 Conclusion

In this paper, we have presented a method to analyze the microservice autoscaling decision-making process using probabilistic model checking, which consists of three phases (i.e., modeling, encoding, and experimentation). For the modeling phase, we have proposed five MDP models to represent the scaling behavior of Kubernetes-based Horizontal Pod Autoscaler. These models consider several factors, which are the CPU utilization, latency, response time, energy consumption, and scalability. For the encoding phase, the PRISM model checker tool has been utilized to encode the models as well as the properties. Then, we have analyzed the models using two quality attributes during the experimentation phase, namely the *Expected Energy Consumption* and *Expected Scalability Value* under two conditions (i.e., high and low). The experimentation aimed to determine which decision models will lead to the least energy consumption of the host with the scalability value nearer to 1. As a result, we have observed that the *Full Model*, which considers all the examined metrics, contributes the lowest energy consumption with the lower bound of its scalability value being nearer to 1, followed by the *Energy Model* and *Scalability Model*, which consider energy consumption and scalability factors, respectively.

The implementation of verification on microservice-based applications has a potential for wider improvements in order to ensure that the existing microservice autoscaling algorithm is truly reliable. The verified policies may also need to consider the cost factors to provide early insights for the developers before deploying the applications in a real infrastructure. Meanwhile, the verified autoscaling may also need to be ranked according to the specific concern before being generated and applied in existing cloud services. Finally, the verified algorithm can also be integrated into the microservice-based application to improve its scaling behavior.

Acknowledgement

Azlan Ismail acknowledges the support of the Fundamental Research Grant Scheme, FRGS/1/2018/ICT01/UITM/02/3, funded by Ministry of Education Malaysia.

Funding

Similar as in the acknowledgement.

Availability of data and materials

Not applicable.

Ethics approval and consent to participate

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Consent for publication

Not applicable.

Authors' contributions

S.N.A.J. conducted the experiments, analyzed the data and wrote the paper; A.I. proposed the idea, reviewed the experiments and revised the paper; V.C. revised the paper. All authors have read and agreed to the published version of the manuscript.

Author details

[1]Institute for Big Data Analytics and Artificial Intelligence (IBDAAI), Kompleks Al-Khawarizmi, Universiti Teknologi MARA (UiTM), 40450, Shah Alam, Selangor, Malaysia. [2]Faculty of Computer and Mathematical Sciences (FSKM), Universiti Teknologi MARA (UiTM), 40450, Shah Alam, Selangor, Malaysia. [3]Department of Civil Engineering and Computer Science Engineering, University of Rome Tor Vergata, 00133, Roma, Italy.

References

1. Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S.: Microservices: The journey so far and challenges ahead. IEEE Software **35**(3), 24–35 (2018)
2. Soldani, J., Tamburri, D.A., Van Den Heuvel, W.-J.: The pains and gains of microservices: A systematic grey literature review. Journal of Systems and Software **146**, 215–232 (2018)
3. Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P.: Elasticity in cloud computing: State of the art and research challenges. IEEE Transactions on Services Computing **11**(2), 430–447 (2018)
4. Bushong, V., Abdelfattah, A.S., Maruf, A.A., Das, D., Lehman, A., Jaroszewski, E., Coffey, M., Cerny, T., Frajtak, K., Tisnovsky, P., *et al.*: On microservice analysis and architecture evolution: A systematic mapping study. Applied Sciences **11**(17), 7856 (2021)
5. Dai, F., Chen, H., Qiang, Z., Liang, Z., Huang, B., Wang, L.: Automatic analysis of complex interactions in microservice systems. Complexity **2020** (2020)
6. McZara, J., Kafle, S., Shin, D.: Modeling and analysis of dependencies between microservices in DevSecOps. In: 2020 IEEE Int'l Conf. on Smart Cloud (SmartCloud), pp. 140–147 (2020). IEEE
7. Chondamrongkul, N., Sun, J., Warren, I.: Automated security analysis for microservice architecture. In: 2020 IEEE Int'l Conf. on Software Architecture Companion (ICSA-C), pp. 79–82 (2020). IEEE
8. Mendonça, N.C., Aderaldo, C.M., Cámara, J., Garlan, D.: Model-based analysis of microservice resiliency patterns. In: 2020 IEEE Int'l Conf. on Software Architecture (ICSA), pp. 114–124 (2020). IEEE
9. Su, G., Liu, L., Zhang, M., Rosenblum, D.: Quantitative verification for monitoring event-streaming systems. IEEE Transactions on Software Engineering **48**, 538–550 (2022)
10. Burroughs, S.: Towards predictive runtime modelling of Kubernetes microservices. PhD thesis, The University of Waikato (2021). https://researchcommons.waikato.ac.nz/handle/10289/14091
11. Ray, K., Banerjee, A.: Prioritized fault recovery strategies for multi-access edge computing using probabilistic model checking. IEEE Transactions on Dependable and Secure Computing (2022)
12. Kochovski, P., Drobintsev, P.D., Stankovski, V.: Formal quality of service assurances, ranking and verification of cloud deployment options with a probabilistic model checking method. Information and Software Technology **109**, 14–25 (2019)
13. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: A model checker for stochastic multi-player games. In: Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 185–191 (2013). Springer

14. Kwan, A., Wong, J., Jacobsen, H.-A., Muthusamy, V.: HyScale: Hybrid and network scaling of dockerized microservices in cloud data centres. In: 2019 IEEE 39th Int'l Conf. on Distributed Computing Systems (ICDCS), pp. 80–90 (2019)

15. Liu, B., Buyya, R., Nadjaran Toosi, A.: A fuzzy-based auto-scaler for web applications in cloud computing environments. In: Int'l Conf. on Service-Oriented Computing, pp. 797–811 (2018). Springer

16. Choi, B., Park, J., Lee, C., Han, D.: pHPA: A proactive autoscaling framework for microservice chain. In: 5th Asia-Pacific Workshop on Networking (APNet), pp. 65–71 (2021). ACM

17. Weyns, D.: An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective. Wiley-IEEE Computer Society Press, NJ, USA (2020)

18. Rossi, F., Cardellini, V., Lo Presti, F.: Hierarchical scaling of microservices in Kubernetes. In: 2020 IEEE Int'l Conf. on Autonomic Computing and Self-Organizing Systems (ACSOS), pp. 28–37 (2020). IEEE

19. Casalicchio, E., Iannucci, S.: The state-of-the-art in container technologies: Application, orchestration and security. Concurrency and Computation: Practice and Experience **32**(17), 5668 (2021)

20. Horizontal Pod Autoscaling (2022). https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

21. Cluster Autoscaler (2022). https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler

22. Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H., Kim, S.: Horizontal pod autoscaling in Kubernetes for elastic container orchestration. Sensors **20**(16), 4621 (2020)

23. Balla, D., Simon, C., Maliosz, M.: Adaptive scaling of Kubernetes pods. In: 2020 IEEE/IFIP Network Operations and Management Symp. (NOMS), pp. 1–5 (2020). IEEE

24. Lehrig, S., Eikerling, H., Becker, S.: Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. In: 11th Int'l ACM SIGSOFT Conf. on Quality of Software Architectures (QoSA), pp. 83–92 (2015)

25. Zhong, Z., Buyya, R.: A cost-efficient container orchestration strategy in Kubernetes-based cloud computing infrastructures with heterogeneous resources. ACM Transactions on Internet Technology **20**(2), 1–24 (2020)

26. Khaleq, A.A., Ra, I.: Intelligent autoscaling of microservices in the cloud for real-time applications. IEEE Access **9**, 35464–35476 (2021)

27. Khazaei, H., Ravichandiran, R., Park, B., Bannazadeh, H., Tizghadam, A., Leon-Garcia, A.: Elascale: Autoscaling and monitoring as a service. In: 27th Ann. Int'l Conf. on Computer Science and Software Engineering (CASCON), pp. 234–240. IBM Corp., USA (2017)

28. Pedram, M.: Energy-efficient datacenters. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **31**(10), 1465–1484 (2012)

29. Hameed, A., Khoshkbarforoushha, A., Ranjan, R., Jayaraman, P.P., Kolodziej, J., Balaji, P., Zeadally, S., Malluhi, Q.M., Tziritas, N., Vishnu, A., *et al.*: A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems. Computing **98**(7), 751–774 (2016)

30. Akhter, N., Othman, M.: Energy aware resource allocation of cloud data center: review and open issues. Cluster Computing **19**(3), 1163–1182 (2016)

31. Beloglazov, A., Abawajy, J., Buyya, R.: Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. Future Generation Computer Systems **28**(5), 755–768 (2012)

32. de Nardin, I.F., da Rosa Righi, R., Lopes, T.R.L., da Costa, C.A., Yeom, H.Y., Köstler, H.: On revisiting energy and performance in microservices applications: A cloud elasticity-driven approach. Parallel Computing **108**, 102858 (2021)

33. Xu, M., Toosi, A.N., Buyya, R.: Ibrownout: An integrated approach for managing energy and brownout in container-based clouds. IEEE Transactions on Sustainable Computing **4**(1), 53–66 (2018)

34. Kwiatkowska, M., Norman, G., Parker, D.: Advances and challenges of probabilistic model checking. In: 2010 48th Ann. Allerton Conf. on Communication, Control, and Computing (Allerton), pp. 1691–1698 (2010). IEEE

35. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Handbook of Model Checking, pp. 963–999. Springer, Cham, Switzerland (2018)

36. Kwiatkowska, M., Parker, D.: Automated verification and strategy synthesis for probabilistic systems. In: Automated Technology for Verification and Analysis, pp. 5–22. Springer, Cham, Switzerland (2013)

37. Naskos, A., Stachtiari, E., Gounaris, A., Katsaros, P., Tsoumakos, D., Konstantinou, I., Sioutas, S.: Cloud elasticity using probabilistic model checking. arXiv preprint arXiv:1405.4699 (2014)

38. Cámara, J., Moreno, G.A., Garlan, D., Schmerl, B.: Analyzing latency-aware self-adaptation using stochastic games and simulations. ACM Transactions on Autonomous and Adaptive Systems **10**(4), 1–28 (2016)

39. Tsai, W.-T., Huang, Y., Shao, Q.: Testing the scalability of SaaS applications. In: 2011 IEEE Int'l Conf. on Service-Oriented Computing and Applications (SOCA), pp. 1–4 (2011). IEEE

40. Moreno, G.A., Cámara, J., Garlan, D., Schmerl, B.: Proactive self-adaptation under uncertainty: A probabilistic model checking approach. In: 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 1–12 (2015)

41. Keefer, D.L.: Certainty equivalents for three-point discrete-distribution approximations. Management Science **40**(6), 760–773 (1994)

42. Jagadeesan, L.J., Mendiratta, V.B.: When failure is (not) an option: Reliability models for microservices architectures. In: 2020 IEEE Int'l Symp. on Software Reliability Engineering Workshops (ISSREW), pp. 19–24 (2020). IEEE