International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Reinforcement Learning for Elasticity Control in Containerized Cloud Applications: A Model-Based Approach with Elastic Docker Swarm

Naimisha Shashikantbhai Trivedi[1], Dhaval Varia[2], Jignesh Harenkumar Vaniya[3] , Chetna Ganesh Chand[4], Nikunj Chunilal Gamit[5]**

Submitted: 20/02/2024 **Revised**: 25/03/2024 **Accepted**: 02/04/2024

Abstract: Software containers are becoming increasingly popular for managing and executing distributed applications on cloud computing resources. By leveraging the horizontal and vertical elasticity of containers "on the fly," workload fluctuations can be accommodated. The majority of current control systems do not consider horizontal and vertical scaling to be interconnected. In this article, we provide Reinforcement Learning (RL) strategies for controlling the vertical and horizontal elasticity of container-based systems to enhance their adaptability to various workloads. Although RL is an interesting technique, it may have a long learning period if nothing is known about the system beforehand. To accelerate learning and discover more effective adaptation strategies, our proposed reinforcement learning approaches—Q-learning, Dyna-Q, and Model-based methods—leverage varying degrees of knowledge about system dynamics. The recommended policies are incorporated into Elastic Docker Swarm, an add-on for the container orchestration platform Docker Swarm. Through prototype-based experiments and simulations, we demonstrate the effectiveness and adaptability of model-based RL techniques.

Keywords: *Elastic Docker Swarm, RL, Dyna-Q, Q-learning, Prototypes, Model-Based, Strategies*

1. Introduction

The cloud is a new kind of computing that allows users to rent out resources according to their own needs, on an as-needed basis. In order to keep up with the ever-changing workload, there are a number of cloud platforms and virtual data centers that provide elastic services. [1, 2] The capacity of a system to dynamically adjust its resources in response to changes in load is known as its elasticity. [3] One of the ever-changing features of cloud computing is this. First, service elasticity improves Quality of Service (QoS) by maximizing certain metrics like response time, CPU load, requests processed per second, etc. Service Level Agreements (SLAs) guarantee quality of service between customers and cloud resource providers. [4, 5] Preventing the system from over-provisioning resources is the second benefit, which lowers total power usage. The term "over provisioning" describes the practice of allocating more resources than are really needed to manage peak demands. [6, 7]

The fundamental aspects of elasticity are efficiency and scalability. How well computer resources are used when scaling is what efficiency is all about. [8, 9] The efficiency improves as the number of resources required decreases. A machine is scalable if and only if it can take on additional work when its resources are added to [10, 11].

When it comes to hosting their services, cloud companies rely on virtualization. Hardware platforms, storage devices, and network resources may all be "virtualized" in the process of producing a virtual version of them. Hypervisors and containers allow for the implementation of virtualization [12, 13]. Containers are a lightweight software alternative to hypervisors that outperform virtual machines in terms of start/stop time and overhead. A software developer may use containers to bundle a programme with all of its dependencies, including libraries, and then distribute it out as a single package. You may do auto scaling in two ways: reactively or proactively [14, 15].

Response time, CPU load, memory use, etc., are all examples of thresholds that may be improved in a reactive strategy. Resources may be raised or lowered depending on the period after which we notice that they have exceeded this threshold. While reactive scaling relies on historical data, proactive scaling uses machine learning or deep learning techniques to forecast future workloads. [16, 17]. Next, we determine how many resources will be required to handle the anticipated workload. Be careful not to scale

[1]*Information Technology Department, Vishwakarma Government Engineering College, Ahmedabad*
ORCID ID: 0000-0002-8395-1586
[2] *Information Technology Department, Vishwakarma Government Engineering College, Ahmedabad*
ORCID ID: 0000-0003-4505-1509
[3] *Information Technology Department, Vishwakarma Government Engineering College, Ahmedabad*
ORCID ID: 0009-0002-4121-0074
[4] *Information Technology Department, Vishwakarma Government Engineering College, Ahmedabad*
ORCID ID: 0009-0003-1876-8472
[5] *Information Technology Department, Vishwakarma Government Engineering College, Ahmedabad*
ORCID ID: 0009-0004-2821-4606
***Corresponding Author Email: naimishatrivedi@gmail.com*

up too quickly while adopting proactive tactics. We have to scale up our resources again soon after we scale down since a rush of requests comes just after we scale down. This is called premature scaling. Premature scaling, then, will result in unnecessary overhead and be ineffective from the perspective of the cloud provider [18, 19].

The adoption of both reactive and proactive strategies in auto-scaling may increase its efficiency [20, 21]. Auto-Regressive Moving Average (ARMA), Auto-Regressive Integrated Moving Average (ARIMA), Moving Average (MA), and other proactive methods may be used to time-series data for the purpose of doing workload prediction. A load balancer may distribute future workloads across many computers. [22, 23] The load balancer dynamically and evenly distributes the burden over all the available nodes by using various load balancing methods. In the long run, this helps the system run better. The term "cloud load balancing" refers to a technique used in cloud computing for dividing up tasks and resources [24].

2. Review of Literature

- When putting their autoscaling rules into practice, Hasan et al. (2022) [26] considered the four threshold values. When autoscaling uses two threshold values, it makes more accurate decisions. In addition to the rule-based autoscaling mechanism, several researchers have proposed an autoscaling mechanism that integrates concepts from control theory. Thanks to a controller, they are now functional. Given a control input, the controller's task is to maintain a specific level of system performance. Systems based on maximum control employ reactive processes.
- Kan, C (2023) [27] When it comes to elastic services, DoCloud offers an auto-scaling platform. Based on the changing workload, it determines the optimal number of containers. Specifically, it scales out using a reactive strategy and scales in the number of containers using a proactive approach when it comes to auto-scaling. By comparing each container's CPU utilization to a threshold value, the threshold-based method is applied in the reactive approach.
- Al-Dhuraibi et al. (2017) [25] offered a rule-based reactive strategy. Using ELASTICDOCKER, this method is applied to the vertical elasticity of containers. In order to adapt the RAM and virtual CPU cores available in containers to different workloads, the ELASTICDOCKER auto-scaler is used. Various experimental values are used to change the upper and lower threshold values of a container. The one with the shortest reaction time is then selected. The writers have fixedly assigned or removed the CPU and RAM from a container. For instance, the auto-scaler adds 256 MB of RAM to the container if the memory utilization figure exceeds the top limit of hits.

3. Objectives

• To create reinforcement learning (RL) systems that can simultaneously regulate the vertical and horizontal elasticity of container-based apps in order to boost adaptability when managing different workloads.

• To investigate the effectiveness of RL solutions that integrate varying degrees of system dynamics information, such as Q-learning, Dyna-Q, and model-based, in order to identify appropriate adaptation policies for container elasticity and to expedite the learning process.

4. Statement of the problem

Effective resource management and scaling are crucial as cloud-native apps developed using containerized microservices gain popularity. Both vertical scaling (resizing computational resources) and horizontal scaling (adding/removing instances) present unique difficulties. In order to maximize performance and minimize resource waste, you must choose when and how to scale in response to changing workload demands. Monitoring various metrics, anticipating future requirements, and promptly allocating resources across the dispersed containerized services are all necessary for putting the proper auto-scaling rules into place. Furthermore, if scaling events are not properly managed, they may impact the operation and availability of applications. The challenge is developing self-adaptive, intelligent scaling systems that react to changing real-world circumstances without compromising cost and SLAs.

5. Significance of the study

The main elements that contribute to optimizing the availability, affordability, and performance of contemporary cloud-based applications are effective resource allocation and scaling. A strong and intelligent auto-scaling capacity is becoming increasingly crucial as more and more companies go to containerized microservice architectures operating in the cloud. Applications must be scalable in order to handle changing demands without under-provisioning, which degrades performance, or over-provisioning, which leaves resources underutilized. Being extremely responsive and elastic is a fundamental requirement for cloud-native applications.

6. Research methodology

- Issue Identification and System Model

We examine a model for applications that is very generic, in which the application is a black-box entity that does certain operations (such as calculation and data access). It is possible to launch many application instances in parallel in order to adequately handle ever-increasing incoming

workloads. Separate instances handle different portions of the incoming requests. The programme reveals its response time requirements, which are stated as a maximum reaction time that should not be exceeded. Application deployment and runtime management may be made easier using software containers, such as Docker.

- Flattening In Both Directions With RL

By interacting with the system in a more natural way, RL methods hope to discover the best way to adapt. The goal of RL strategies is to minimize a numerical cost signal by learning what to do (i.e., by mapping circumstances to actions). The trade-off between exploring and exploiting is one of the problems that comes up in RL. An RL agent's goal is to minimize the gained cost by giving preference to previously attempted actions that were successful (exploitation). The agent uses an approximation of the so-called Q-function to minimize the anticipated long-term cost. The projected long-term cost that follows the execution of action an in state s is represented by the Q(s,a) terms. To scale, the agent uses the Q-function to choose which action to take: given a system state s, it takes action a that minimizes Q(s,a). The scaling strategy is improved by updating Q(s,a) over time based on the actual incurred expenses. In order to predict the long-term cost, the various RL methods use different methodologies.

The condition of the programme at time i is defined as si = (ki,ui,ci), where ki is the quantity of containers (application instances), ui is the utilization of the CPU, and ci is the proportion of CPU allocated to each container. By "S," we mean the set of allui) states that the application makes use of. Although u¯ and c¯u are appropriate quanta, we discretize them despite the fact that CPU utilization (and CPU share, ci) are actual numbers. The number of containers where Kmax is the maximum application replication degree is denoted by ci and the set {0,u,...,L¯ u¯} is also assumed.

Here, A is the set of all actions, and for any state s ϵ S, we have a set of possible adaptation actions A(s) ⊆ A. The 5-action model and the 9-action model are the two alternatives that we provide. Both horizontal and vertical scaling are possible in the 5-action model, but in the 9-action model, we may scale in both directions at once. In the formal sense, the 5-action model is represented by A = {-r, - 1, 0, 1, r}, where ±r denotes a vertical scaling (i.e., +r to increase CPU share and r to remove CPU share), ±1 denotes a horizontal scaling (i.e., +1 to scale-out and 1 to scale-in), and a = 0 denotes the do nothing choice. A = {1, 0, +1} x {r, 0, r} is another way of looking at the 9-action model. In a 5-action model, for example, the accessible actions in state s with k = Kmax and c = Mc¯ are A(s) = {-r, - 1, 0} (i.e., we cannot execute additional scale up and out operations), although obviously not all actions are available in every application state.

$$
\begin{aligned}
c(s,a,s') &= w_{\text{adp}} \frac{\mathbb{1}_{\{vertical-scaling\}} c_{\text{adp}}}{c_{\text{adp}}} + \\
&\quad + w_{\text{perf}} \frac{\mathbb{1}_{\{R(k+a_1,u',c+a_2)>R_{\max}\}} c_{\text{perf}}}{c_{\text{perf}}} + \\
&\quad + w_{\text{res}} \frac{(k+a_1)(c+a_2)c_{\text{res}}}{K_{\max} \cdot c_{\text{res}}} \\
&= w_{\text{adp}} \mathbb{1}_{\{vertical-scaling\}} + \\
&\quad + w_{\text{perf}} \mathbb{1}_{\{R(k+a_1,u',c+a_2)>R_{\max}\}} + \\
&\quad + w_{\text{res}} \frac{(k+a_1)(c+a_2)}{K_{\max}}
\end{aligned}
$$

Algorithm 1 Dyna-Q

1: Initialize $Q(s,a)$ and $Model(s,a)$, $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$
2: **while** true **do**
3: $s \leftarrow$ observe the application state
4: $a \leftarrow$ select an action using the current estimate of Q
5: Observe the next state s' and the incurred cost c
6: Update $Q(s,a)$ using Eq. 2
7: $Model(s,a) \leftarrow c, s'$
8: **for** $i = 0 \rightarrow n$ **do**
9: $s \leftarrow$ random state previously observed
10: $a \leftarrow$ random action previously taken in s
11: $c, s' \leftarrow Model(s,a)$
12: Update $Q(s,a)$ using Eq. 2
13: **end for**
14: **end while**

$$
Q(s,a) = \sum_{s' \in \mathcal{S}} p(s'|s,a) \left[c(s,a,s') + \gamma \min_{a' \in \mathcal{A}(s')} Q(s',a') \right] \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)
$$

$$
w_{\text{adp}}, \; w_{\text{perf}} \text{ and } w_{\text{res}},
$$

where 1{•} is the indicator function and sum of all is 1.

The application's reaction time in the state s = (k, u, c), and are non-negative weights for the various expenses. In addition, we break down action an into its component parts, container count (a1) and CPU share (a2), in that order.

We take a look at three distinct RL methods, each with its own set of assumptions and learning process. We start with the easy-to-understand Q-learning method; it doesn't rely on models and doesn't need understanding the dynamics of the system. Next, we introduce Dyna-Q, a system model builder that uses real-world data. In addition, we provide a model-based method that updates the Q-function depending on the known (or estimated) system dynamics. By providing RL agents with a system model, the model-based approach accelerates the learning phase via exploratory activities.

A. Q-learning

By averaging its samples, Q-learning is able to approximate the ideal Q-function, Q*. With a probability of ϵ, Q-learning chooses a random action at decision step i to improve its application knowledge; with a probability of 1 - ϵ, it chooses the greedy action by exploiting its application knowledge (i.e., an I = arg min). In this paper, we examine the simple ϵ -greedy action selection method $_{\cdot a \in A(s_i)} Q(s_i, a))$. The greedy policy typically chooses the best known action for a given state, but it prefers to

explore less optimum choices with low probability. Here is how Q(si, ai) is changed at the conclusion of each time period i:

$$Q(s_i, a_i) \leftarrow (1-\alpha)Q(s_i, a_i) + \alpha \left[c_i + \gamma \min_{a' \in \mathcal{A}(s_{i+1})} Q(s_{i+1}, a') \right]$$

where $\alpha \in 2$ [0, 1] is the parameter for the learning rate, and is the range of values for the discount factor. Keep in mind that (2) only incorporates the newly observed variables into the previous estimate of Q, such as the cost ci and the discounted cost anticipated when the system is in si+1, min $\min_{a' \in \mathcal{A}(s_{i+1})} Q(s_{i+1}, a')$.

B. Dyna-Q

Dyna-Q is an alternative to Q-learning that attempts to mimic the system's interaction with its surroundings in order to expedite the learning process. The Dyna-Q learning is summarized in Algorithm 1. Dyna-Q, like Q-learning, watches the application state at runtime and chooses an adaptation action based on the estimations of Q(s, a). Dyna-Q uses a sampling system model, Model(s, a), to mimic the application-environment interaction after time step i concludes (lines 8–13). Dyna-Q keeps the investigated state-action pair (s, a) up-to-date at runtime by storing the next state s0 and cost c; for details, see line 7. Under the assumption of a deterministic environment, Dyna-Q utilizes the state-action pairs previously recorded and updates the Q-function using (2).

- Model-Based Reinforcement Learning

Thirdly, we look at an RL technique that relies on a complete backup model. The complete backup method calculates the Q-function using the Bellman equation in conjunction with a potentially approximated model of the system:

We substitute the unknown cost function and the unknown transition probabilities p(s0 | s, a). $\hat{c}(s, a, s'), \forall s, s' \in \mathcal{S}$

based on the estimations they made using their knowledge. In order to determine $p(s'|s,a)$, Simply estimating the probability of CPU utilization transitions will do the trick $P[u_{i+1} = u' | u_i = u]$. Actually, it is worth noting that:

$$p(s'|s,a) = P[s_{i+1} = (k', u', c') | s_i = (k, u, c), a_i = a] = \begin{cases} P[u_{i+1} = u' | u_i = u] & k' = k + a_1 \wedge \\ & \wedge\, c' = c + a_2 \\ 0 & \text{otherwise} \end{cases}$$

in which the scaling action, denoted as a = (a1, a2), is specified in terms of the updated amount of CPU share (a2) and the updated number of containers (a1). The fact that u is a discrete set value means that we may say $P_{j,j'} = P[u_{i+1} = j' \bar{u} | u_i = j\bar{u}],\ j, j' \in \{0, \ldots, L\}$ "for short" in this context. In the given timeframe, ni,jj0 represents the number of occurrences when the CPU utilization switches from state ju¯ to j0 u¯. $\{1, \ldots, i\}, j, j' \in \{0, \ldots, L\}$ At moment i, the estimated probabilities of the transition are $\widehat{P_{j,j'}} = n_{i,jj'} / \sum_{l=0}^{L} n_{i,jl}$, along with estimating pˆ(s0 |s, a). The total of two terms, the known cost and the unknown cost, may be expressed as the estimations of the immediate cost c(s, a, s0):

$$c(s, a, s') = c_k(s, a) + c_u(s')$$

In this situation, the known cost ck(s, a) takes into consideration the costs of adaptation and resources, but it is action and state dependent. What happens in the following state s0 determines the unknown cost cu(s0). The performance penalty is taken into consideration by cu(s0). We need to make an online estimate of cu(s0) as we are assuming that the application model is unknown. Hence, the RL agent assesses cu,i (s0) at time i after seeing the immediate cost ci:

$$c_{u,i}(s') = c_i - c_{k,i}(s, a)$$

Algorithm 2 Model-Based Reinforcement Learning Update

1: Update estimates $\widehat{P_{j,j'}}$ and $\hat{c}_{u,i}(s_i)$
2: **for all** $s \in \mathcal{S}$ **do**
3: **for all** $a \in \mathcal{A}(s)$ **do**
4: $Q(s,a) \leftarrow \sum_{s' \in \mathcal{S}} \hat{p}(s'|s,a) \cdot$
5: $\cdot \left[\hat{c}(s,a,s') + \gamma \min_{a' \in \mathcal{A}(s')} Q(s', a') \right]$
6: **end for**
7: **end for**

In order to revise our estimate of the unknown cost cˇu,i(s0), we follow these steps:

$$\hat{c}_{u,i}(s') \leftarrow (1-\alpha)\hat{c}_{u,i-1}(s') + \alpha c_{u,i}(s')$$

The projected cost for the undetermined $\hat{c}_{u,i}(s'$ is then used to calculate the expense of implementing an in s in accordance with (5). In the following state s0 = (k0, u0, c0), it may be heuristically assumed that the predicted cost due to Rmax violation does not decrease if the number of containers, CPU utilization, and/or CPU share are lowered, given a state e s = (k, u, c). The converse is also true. Consequently, when $\hat{c}_{u,i}(s), \forall s \in \mathcal{S}$.

Here are several qualities that can be enforced:

$$\hat{c}_{u,i}(s) \leq \hat{c}_{u,i}(s') \qquad \forall k \geq k', u \leq u' c \geq c'$$
$$\hat{c}_{u,i}(s) \geq \hat{c}_{u,i}(s') \qquad \forall k \leq k', u \geq u', c \leq c'$$

- Docker-Based Technology

Containerized applications may be easily created, deployed, and managed using Docker, an open-source platform. A Docker container is an instance of a container snapshot (or image), which includes the programme combined with all the data required for its execution (e.g., dependencies, configuration file). You may construct and execute containers with Docker using REST APIs or a command-line interface. The Docker Engine is part of Docker. With Docker, you may set a container's resource

quota, which controls how much of the hosting machine's CPU and RAM the container an consume. Vertical elasticity is achieved by being able to change the resource quota during runtime. Starting with version 1.12, Docker integrated the swarm mode with the Docker Engine, making it easy to assign numerous containers on distributed computing resources.

Elastic Docker Swarm Architecture

Here are the primary parts that make up the system:

 Client: This is the client application or user interface that starts the requests or actions in the system.

 Message Broker: It is the main communication hub that enables the messaging between different components of the system to be done asynchronously. It is a go-between that enables components to send and receive messages without direct point-to-point connections.

 Docker Monitors: These parts are in charge of overseeing and controlling the Docker containers or containerized applications. They interact with the Message Broker to get instructions or notifications and communicate with Nodes to do the actions related to containers.

 Nodes: These are the resources or servers where Docker containers are deployed and executed. Nodes talk to the Docker Monitors and the Message Broker to get instructions and to give feedback on container operations.

 Container Manager: This component is in charge of the containers which are spread across the Nodes. It talks to the Message Broker to get requests or instructions and communicates with the Nodes to do container-related actions like deployment, scaling, or load balancing.

The flow of communication and interactions in the system can be described as follows:

1. The Client starts a request or an action, which is forwarded to the Message Broker.

2. The Message Broker directs the request or action to the right components like Docker Monitors, Nodes, or the Container Manager, according to the message content or topic.

3. The Docker Monitors get messages from the Message Broker and communicate with Nodes to execute container-related operations, like starting, stopping, or monitoring containers.

4. The Nodes carry out the operations requested by the containers and inform the Docker Monitors and the Message Broker with the status updates or feedback.

5. The Container Manager gets the messages from the Message Broker and coordinates container management tasks across several Nodes, for instance, deploying new containers, scaling the existing ones or performing load balancing.

6. The Container Manager talks to Nodes to carry out container management tasks and gets feedback or status updates from them.

7. The system architecture supports the separation of communication between components, scalability, and the ability to manage and orchestrate containerized applications across multiple compute resources.

• Elastic Docker Swarm

Elastic Docker Swarm, or EDS for short, is what we propose as a way to provide Docker robotic capabilities. It adds the MAPE control loop by extending the Docker Swarm architecture. The self-adaptation functions are handled by the latter, which consists of four primary components: monitor, analyze, plan, and execute. Application and execution environment data is collected by the Monitor. To find out whether an adjustment is helpful, the Analyze component looks at the data that was obtained. If the adaptation is necessary, the application's adaptation strategy is determined in the strategy component and then executed in the Execute component. We were able to seamlessly include our MAPE components into Docker due to its modular design and extensive API support.

7. Results and Discussion

In this work, we use simulation and EDS-based experiments to assess the suggested RL methods in depth.

Simulation

We begin by contrasting the model-free method with the model-based RL techniques. Secondly, we look at the advantages of a 5-action model and a 9-action model. Thirdly, we demonstrate the adaptability of RL methods in learning multiple adaptation strategies to prevent Rmax violations, resource waste, or frequent adaptations, depending on the weights of the cost functions. Since it is reasonable to assume that the application gets M independent and random requests, D deterministic service time, and ki containers utilized at time step i equals the number of servers, we model the reference application as an M/D/ki queue. The maximum goal response time (Rmax) is 50 milliseconds, and the service rate (μ) is 200 times the number of requests per second (ci), where ci is the amount of CPU share (ϵ = 0 to 1].

Here are the parameters that are used by the RL algorithms: For Q-learning and Dyna-Q, the discount factor Y is 0.99, the learning rate ↵ is 0.1, and ϵ is equal to 1/I, where i is the time taken for the simulation. The application state is discretized using $\bar{u}$ = 0.1 and $\bar{c}$ = 10%. An Intel Core i7-4700MQ (8 cores @ 2.40 GHz) and 8 GB of RAM are the specifications of the system that operates the simulation. Table details the outcomes of the

9-action model's simulation, whereas Table details the outcomes for the 5-action model.

For the given set of weights (wperf = 0.09, wres = 0.90, and wadp = 0.0101), it is crucial to prevent R max violations. We can see that the 9-action model often slows down learning by comparing Tables.

A conclusion might elaborate on the importance of the work or suggest applications and extensions.

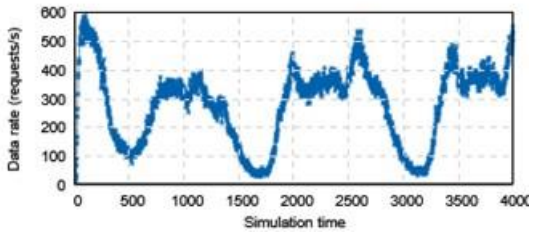

Fig. 1: Application workload used in simulation.

Table 1: Performance of the application with several configurations of RL policies and weights of the cost function, as analysed via simulation using the 5-action adaption model.

Weights	**w_{perf} = 0.09, w_{res} = 0.90, and w_{adp} =0.01**			**w_{perf} = 0.09, w_{res} = 0.90, and w_{adp} = 0.01**			**w_{perf} = w_{res} = w_{adp} = 0.33**		
Policy	**Q-learning**	**Dyna-Q**	**Model-based**	**Q-learning**	**Dyna-Q**	**Model-based**	**Q-learning**	**Dyna-Q**	**Model-based**
R_{max} violations (%)	17.87	7.12	2.37	46.16	56.64	99.8	19.32	19.52	17.17
Average CPU utilization (%)	55.83	48.66	60.54	72.63	79.83	99.85	55.89	53.68	69.01
Average CPU share (%)	62.84	80.21	87.62	54.34	53.79	11.01	68.5	73.23	86.12
Average number of containers	4.49	3.88	2.53	3.49	2.95	1.09	4.28	3.95	2.48
Median R (ms)	13.57	8.97	10.39	35.66	1	1	11.6	9.41	12.04

Table 2: Simulation-based analysis: Application performance under different configurations of cost function weights and RLpolicies, when the 9-action adaptation model is used.

Weights	**w_{perf} = 0.09, w_{res} = 0.90, and w_{adp} = 0.01**			**w_{perf} = 0.09, w_{res} = 0.90, and w_{adp} = 0.01**			**w_{perf} = w_{res} = w_{adp} = 0.33**		
Policy	**Q-learning**	**Dyna-Q**	**Model-based**	**Q-learning**	**Dyna-Q**	**Model-based**	**Q-learning**	**Dyna-Q**	**Model-based**
R_{max} violations (%)	27.17	25.77	2.85	61.18	62.58	99.8	39.69	35.64	19.5
Average CPU utilization (%)	62.82	63.39	60.73	80.95	81.89	99.85	69.22	65.08	70.75
Average CPU share (%)	61.32	62.6	87.43	46.62	46.32	11.04	53.62	54.61	78.35
Average number of containers	3.87	3.56	2.57	3.08	3.7	1.09	3.89	4.35	2.56
Median R (ms)	15.59	15.29	10.15	1	1	1	25	20.53	15.16
Adaptations (%)	88.98	92.53	37.32	89.38	94.3	2.95	85.7	91.1	40.29

In this third scenario, we strike a compromise between the three deployment objectives, with w_{perf} = w_{res} = w_{adp} = 0.33. In comparison to the 9-action model , the 5-action model is able to decrease the amount of R_{max} violations and adaptations. Compared to 9 actions, Q-learning and Dyna-Q have 19% R_{max} violations, but they underutilize computational resources, with an average utilization of 54-56% instead of 65-69%. In this scenario as well, the model-based solution learns a more effective adaption approach, the impact of which is negligible relative to the action count. R_{max} violations are reduced to 17% with 5 actions of the model-based approach, which achieves an average resource utilization of 69%. It typically uses 2.5 containers to operate the programme, with each container having the ability to consume 86% of the given CPU.

Prototype-based

Motivated by the promising outcomes in the simulation, we test the RL algorithms in a real-world setting. We achieve this goal by incorporating the suggested policies into EDS. Specifically, we evaluate Q-learning in comparison to the model-based RL solution for both the 5-action and 9-action models. We do not take Dyna-Q into account because of space constraints, even though it offers a little improvement over Q-learning. Large Amazon Elastic Compute Cloud instances, each with two virtual CPUs and eight gigabytes of RAM, are used to install EDS. At user request, the reference app calculates the sum of the first n Fibonacci numbers (complexity O(n2)).

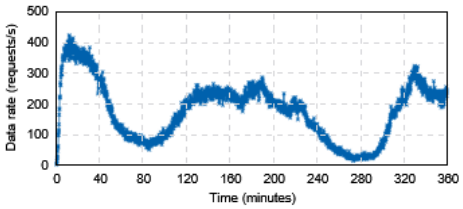

Fig. 2: Workload used in the prototype-based experiments

Table 3: Application performance in various combinations of cost function weights and RL policies, using the 5-action adaptation model, as shown in prototype-based studies.

Weights	**w_{perf} = 0.09, w_{res} = 0.90, and w_{adp} = 0.01**		**w_{perf} = 0.09, w_{res} = 0.90, and w_{adp} = 0.01**		**w_{perf} = w_{res} = w_{adp} = 0.33**	
Policy	**Q-learning**	**Model-based**	**Q-learning**	**Model-based**	**Q-learning**	**Model-based**
R_{max} violations (%)	29.77	12.1	38.21	97.81	55.06	39.23
Average CPU utilization (%)	48.81	37.35	50.87	90.85	63.55	52.22
Average CPU share (%)	88.7	47.34	89.19	24.01	84.43	52
Average number of containers	1.7	5.11	1.54	2.12	1.48	4.21
Median R (ms)	4.18	6.11	4.24	25959.45	223.35	8.3
Adaptations (%)	65.65	50.81	67.48	61.31	66.46	32.31

Table 4: Experimental setup for the 9-action adaptation model; application performance tested with varying weights for the cost function and RL rules.

Weights	**wperf = 0.09, wres = 0.90, and wadp = 0.01**		**wperf = 0.09, wres = 0.90, and wadp = 0.01**		**wperf = wres = wadp = 0.33**	
Policy	**Q-learning**	**Model-based**	**Q-learning**	**Model-based**	**Q-learning**	**Model-based**
Rmax violations (%)	80.54	24.11	88.19	97.58	96.77	33.6
Average CPU utilization (%)	82.21	31.53	87.29	91.05	94.07	60.19
Average CPU share (%)	45.14	46.03	39.37	20.16	40.4	36.72
Average number of containers	1.58	7.1	1.46	2.32	1.49	4.59
Median R (ms)	9738.49	6.92	16587.03	29654.97	18740.43	18.63
Adaptations (%)	93.51	44.68	90.55	59.68	88.71	28

We begin with the following set of weights: w_{perf}=0.90, w_{res}=0.09, w_{adp}= 0.01. In this scenario, minimizing adaptation costs is more essential than optimizing application reaction time, and conserving resources is secondary. Adapting the application deployment is something that Q-learning gradually becomes better at. Table show that when the 5-action model is taken into account, Q-learning often alters the application deployment, carrying out horizontal and vertical scaling operations 66% of the time. In addition, 30% of the time, the application's reaction time is longer than R_{max}. The model-based approach uses the system knowledge to bring the number of R_{max} violations down to 12%. In contrast to Q-learning, which makes use of a smaller number of containers, the model-based strategy deploys 5.11 containers, each of which can access 47% of the CPU, rather than 1.7 containers, which can access 89% of the CPU.

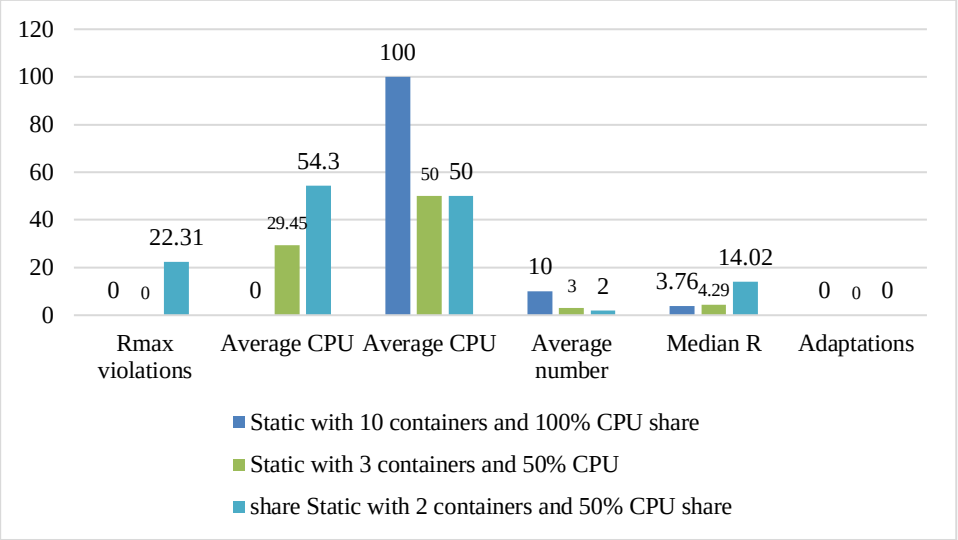

Fig. 3: Prototype-based experiments: Application performance resulting from different configurations of static deployment

Discussion

The results of these trials highlight the value of system information in enhancing the learning process. Therefore, the model-based approach just estimates the unknown system dynamics based on the experience. Curiously, it achieves top performance in all scenarios that are taken into consideration. Additionally, the results demonstrate that the learning task is slowed down by the 9-action model due to the increased combination of state-actions that need to be assessed. But the model-based strategy succeeds because it makes full use of the 9-action model to scale horizontally and vertically at the same time while minimizing modifications. Experiments with prototypes often corroborate the merits of the model-based approach. In comparison to a static setup, as shown in Table," the advantages of a model-based RL algorithm may be shown in Tables. Static configurations may achieve certain edge-case deployment objectives in terms of performance, but they are application-specific and hence unable to handle unexpected spikes or changes in demand. On the other hand, the RL-based method is adaptable and flexible; all that's needed is to define the deployment goals. [28,29]. The difficulty of the computations that follow is $O(|\mathcal{S}|^2|\mathcal{A}|)$ However, the complexity decreases to due to the small number of actions and the fact that many transition probabilities are equal to 0. $O(K_{\max}\lceil\frac{1}{\bar{c}}\rceil\lceil\frac{1}{\bar{u}}\rceil^2)$.[30]

8. Conclusion

The majority of current elasticity rules rely on heuristics that are based on thresholds and need the specification of precise ways to accomplish objectives. A variety of RL strategies for managing the elasticity of container-based applications are presented in this article with the goal of developing a more generic and adaptable solution. In particular, we have developed and tested model-based and model-free approaches, which make use of varying degrees of information on the dynamics of the system. The various RL techniques were evaluated in depth via the use of simulations and tests based on prototypes. A self-adapting extension of Docker Swarm, we call EDS, was born out of this need.

8.1 Findings of the study

The experiments showed the significance of using system knowledge to make RL for container elasticity better. The model-based RL solution, which is based on experience, used to estimate the unknown system dynamics, got the highest performance among all the scenarios by successfully combining the vertical and horizontal scaling

to reduce the adaptations. The prototype experiments proved that the model-based approach is better than the static configurations that cannot adapt to the changes in the workload. Although the model-based solution is more computationally demanding, its ability to dynamically learn the best adaptation strategy that aligns with the user-specified deployment objectives makes it a promising approach for managing the elasticity of container-based applications with varying workloads.

8.2 Scope for further research

The RL principles that have been suggested to manage applications with several components, such as microservices, that are set up in a geo-distributed fog computing setting. In this kind of setup, the computing resources are heterogeneous and the network latencies are not insignificant. A new decentralized control pattern for managing the elasticity of container-based systems, such as returning to a hierarchical control solution, will be designed and integrated into the EDS architecture.

Conflicts of interest

The authors declare no conflicts of interest.

References

[1] Zhang, Q., Cheng, L., Boutaba, R. (2020). Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications, 1(1): 7-18.

[2] Smith, J., Nair, R. (2018). Virtual Machines: Versatile Platforms for Systems and Processes. Elsevier.

[3] Seo, K.T., Hwang, H.S., Moon, I.Y., Kwon, O.Y., Kim, B.J. (2019). Performance comparison analysis of Linux container and virtual machine for building cloud. Advanced Science and Technology Letters, 66(105-111). https:// doi.org/10.14257/ASTL.2014.66.25

[4] Moreno-Vozmediano, R., Montero, R.S., Huedo, E., Llorente, I.M. (2019). Efficient resource provisioning for elastic Cloud services based on machine learning techniques. Journal of Cloud Computing, 8(1): 5. https://doi.org/10.1186/s13677-019-0128-9

[5] Jiang, J., Lu, J., Zhang, G., Long, G. (2017). Optimal cloud resource auto-scaling for web applications. In 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, Delft, Netherlands, pp. 58- 65. https://doi.org/10.1109/CCGrid.2013.73

[6] Roy, N., Dubey, A., Gokhale, A. (2021). Efficient autoscaling in the cloud using predictive models for workload forecasting. 2011 IEEE 4th International Conference on Cloud Computing, Washington, DC, USA.

[7] Messias, V.R., Estrella, J.C., Ehlers, R., Santana, M.J., Santana, R.C., Reiff-Marganiec, S. (2016). Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure. Neural Computing and Applications, 27(8): 2383-2406. https://doi.org/10.1007/s00521-015- 2133-3

[8] Cocaña-Fernández, A., Sánchez, L., Ranilla, J. (2016). Leveraging a predictive model of the workload for intelligent slot allocation schemes in energy-efficient HPC clusters. Engineering Applications of Artificial Intelligence, 48: 95-105.

[9] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P. (2017). Elasticity in cloud computing: state of the art and research challenges. IEEE Transactions on Services Computing, 11(2): 430-444. https://doi.org/10.1109/TSC.2017.2711009

[10] Moreno-Vozmediano, R., Montero, R.S., Llorente, I.M. (2022). IaaS cloud architecture: From virtualized datacenters to federated cloud infrastructures. Computer, 45(12): 65-72. https://doi.org/10.1109/MC.2012.76

[11] De Abranches, M.C., Solis, P. (2023). An algorithm based on response time and traffic demands to scale containers on a Cloud Computing system. In 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), pp. 343-350.

[12] Padala, P., Hou, K.Y., Shin, K.G., Zhu, X., Uysal, M., Wang, Z., Merchant, A. (2019). Automated control of multiple virtualized resources. In Proceedings of the 4th ACM European Conference on Computer Systems, pp. 13-26. https://doi.org/10.1145/1519065.1519068

[13] Gao, Y., Li, Q. (2019). A new framework for the complex system's simulation and analysis. Cluster Computing, 22: 9097-9104. https://doi.org/10.1007/s10586-018-2071-9

[14] GOOGLE. Google horizontal pod auto-scaler. Available from https: //github.com /kubernetes/kubernetes/blob/release-1.2/docs/design/ horizontal-pod-autoscaler.md, accessed on dated 10-05- 2019

[15] Meng, Y., Rao, R.N., Zhang X., Hong, P. (2017). CRUPA: A container resource utilization prediction algorithm for auto-scaling based on time series analysis. In: 2016 International Conference on Progress in Informatics and Computing (PIC), pp 468-472.

[16] E. A. Brewer, (2018)"Kubernetes and the path to cloud native," in Proceedings of the Sixth ACM Symposium on Cloud Computing, p. 167, Kohala, HI, USA.
[17] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P. (2017). Autonomic vertical elasticity of docker containers with ELASTICDOCKER. In 2017 IEEE 10th international conference on cloud computing (CLOUD), Honolulu, CA, USA, pp. 472-479.
[18] Hasan, M.Z., Magana, E., Clemm, A., Tucker, L., Gudreddi, S.L.D. (2022). Integrated and autonomic cloud resource scaling. In 2012 IEEE Network Operations and Management Symposium, Maui, HI, USA, pp. 1327-1334.
[19] Kan, C. (2023). DoCloud: An elastic cloud platform for Web applications based on Docker. 2016 18th International Conference on Advanced Communication Technology (ICACT), Pyeongchang, South Korea.
[20] Z. Zhong and R. Buyya, (2020) "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," ACM Transactions on Internet Technology, vol. 20, no. 2, pp. 1–2.
[21] T. Menouer,(2020) "KCSS: Kubernetes container scheduling strategy," 1e Journal of Supercomputing, pp. 1–2.
[22] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, (2022)"Horizontal pod autoscaling in Kubernetes for elastic container orchestration," Sensors, vol. 20, no. 16, p. 4621.