Can Reinforcement Learning be Generalized for Efficient Auto-Scaling in Containerized Clouds?

José Santos*, Efstratios Reppas†, Tim Wauters*, Bruno Volckaert*, Filip De Turck*
* IDLab, Department of Information Technology at Ghent University - imec, 9000 Ghent, Belgium
Email: {josepedro.pereiradossantos, tim.wauters; bruno.volckaert, filip.deturck}@UGent.be
† National Technical University of Athens (NTUA), Greece

***Abstract*—The rapid adoption of containerized cloud environments requires robust and efficient Auto-Scaling (AS) mechanisms to ensure adequate resource utilization, high performance, and cost-effectiveness. Traditional AS approaches, often based on predefined thresholds, fail to adapt well to dynamic workloads. This paper investigates the potential of Reinforcement Learning (RL) as a generalized solution for efficient AS in containerized clouds. Building on previous studies, this paper examines whether RL approaches can learn adaptive scaling policies when trained on diverse workload datasets and tested across different scenarios. A Multi-Objective (MO) reward function has been designed to optimize key performance factors such as the application's response time, and resource utilization. The results demonstrate that RL algorithms can effectively balance competing objectives and adapt to changing workloads. The *Latency* strategy resulted in lower latency but required more pods (7.4) and slightly higher CPU usage (28.92%). In contrast, the *Cost* strategy minimized deployment costs with fewer pods (3.56) and lower CPU usage (24.45%). This study highlights the versatility and efficiency of RL in managing complex, real-time scaling decisions in containerized cloud infrastructures.**

***Index Terms*—Auto-scaling, Containers, Next-generation networks, Cloud-native, Orchestration, Reinforcement Learning**

I. Introduction

The proliferation of containerized cloud environments has transformed the landscape of modern computing, offering unprecedented levels of scalability, flexibility, and efficiency [1]. Containers encapsulate applications and their dependencies, enabling consistent deployment across various environments and simplifying resource management [2]. However, the dynamic nature of cloud workloads poses significant challenges for maintaining optimal performance and cost-efficiency. Traditional Auto-Scaling (AS) mechanisms, reliant on static thresholds and heuristic-based rules, often struggle to respond effectively to fluctuating demands and diverse workload patterns [3].

AS mechanisms are crucial for cloud infrastructure management to dynamically adjust resource allocation to meet performance targets while minimizing operational costs. Effective strategies can lead to significant cost savings and improved user experiences by ensuring that applications receive the necessary resources during peak loads and scale down during periods of low demand [4]. Despite recent advancements, existing AS approaches [5]–[7] often lack the adaptability required to handle the complexity of modern cloud environments. With the rise of distributed paradigms such as Fog Computing [8] and Edge Computing [9], more flexible and responsive AS strategies are needed to improve performance, and manage computing resources efficiently in these highly distributed environments. In recent years, Reinforcement Learning (RL) has emerged as a promising approach in network management challenges [10]. By formulating the problem as a sequential decision-making process, RL enables the development of adaptive policies that can learn from and respond to varying workload conditions. In contrast to traditional methods, RL does not rely on predefined thresholds, but instead learns optimal actions through interactions with the environment, producing a learned policy well-suited for dynamic workload scenarios.

This paper explores the application of RL for the AS of containerized applications, with a strong emphasis on the following research question: *Can Reinforcement Learning (RL) be generalized for efficient auto-scaling in containerized clouds?* This work investigates whether RL algorithms, especially based on the DeepSets (DS) Neural Network (NN) architecture [11] (detailed in Sec. III), can generalize across different workload datasets and adapt to new, unseen workloads, thereby providing a robust solution for dynamic resource management.

In contrast to our prior work [12], [13], a Multi-Objective (MO) reward function has been designed to consider multiple key performance indicators such as the application's response time, the overall deployment cost, and the container's resource utilization. By training RL algorithms on diverse datasets and testing their performance in various workload scenarios, this paper aims to demonstrate the potential of RL to enhance the efficiency and generalizable capability of AS mechanisms. The main contributions of this paper are twofold:

- **Novel RL design**: Sec. III presents the refined RL design, including observation state, action space, and the MO reward function. This approach considers three performance factors, including deployment cost, the application's response time, and resource efficiency.
- **Performance Evaluation with Microservice Benchmarks**: The evaluation considered two microservice-based applications: *1)* a database application named as Redis Cluster (RC) [14], and *2)* a multi-tier web application denoted as Online Boutique (OB) [15]. Experiments show that the presented RL approach can generalize well for OB when trained on a smaller application as RC, demonstrating the versatility of the DS NN (Sec. V).

Results show superior performance compared to previous work [12].

The remainder of this paper is organized as follows: Sec. II reviews recent studies in the field of AS in cloud computing focused on Machine Learning (ML) and RL methodologies. Sec. III details the applied methodology, including the design of the RL approach and the MO reward function. Sec. IV presents the experimental setup and Sec. V presents the obtained results while discussing the strengths and limitations of RL. Finally, Sec. VI concludes the paper and outlines directions for future research.

II. Related Work

ML-based techniques are becoming increasingly prevalent in modern AS strategies due to their increased efficiency and adaptability [16]–[18]. Most of these methodologies aim to build models that accurately estimate resource needs under specific workloads [19]–[21]. These are particularly robust to dynamic demands, as they can adjust model parameters in real-time in response to notable events (i.e., online learning). While offline training is possible, it often requires significant human intervention, which diminishes the main advantage of these algorithms. Google Autopilot [22] is an example of an ML-based approach that automatically configures resources by adjusting the number of concurrent tasks in a job (horizontal scaling) and the CPU/RAM limits for individual tasks (vertical scaling). Autopilot minimizes the gap between resource limits and actual resource usage, thus reducing the risk of Out of Memory (OOM) errors and performance degradation due to CPU throttling. By applying ML algorithms to analyze historical data and discern patterns from past task executions, Autopilot has demonstrated significant reductions in resource utilization and OOM errors.

However, a major drawback of these ML-based approaches is the high execution time required to converge to a stable model, which can result in suboptimal performance during the learning phase. Recent efforts focus on optimizing execution time while ensuring the accuracy and stability of the resulting ML models. The potential of ML-based AS strategies to revolutionize application performance makes this an active and promising research field for future cloud computing environments. Leveraging insights from our earlier work [12], we have designed a MO reward function that considers multiple key performance indicators such as the overall deployment cost, the application's response time, and container resource utilization. By training RL algorithms on diverse datasets and testing their performance in various workload scenarios, this paper aims to demonstrate the potential of RL to enhance the efficiency and generalizability of AS mechanisms.

III. Reinforcement Learning (RL) Methodology

The *gym-hpa* framework[1] [12] has been developed to enable scalable and cost-effective training of RL algorithms for AS tasks in the Kubernetes (K8s) platform. Traditional

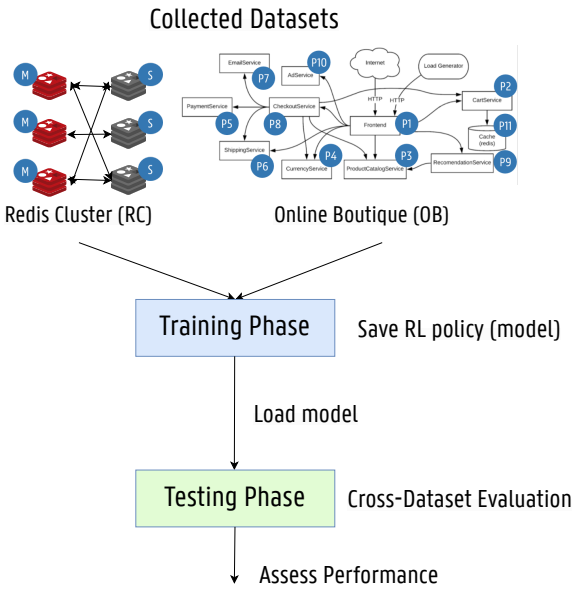


Fig. 1: A schematic overview of the applied methodology. Cross-dataset evaluations are set up based on the *gym-hpa* framework and the DS-based RL algorithms.

OpenAI Gym-based environments are typically designed to speed up the training process of RL. As such, our *gym-hpa* framework supports multiple RL algorithms focused on generating an AS strategy using input from pre-collected datasets[2] or by operating in an online mode, where training occurs in a live K8s cluster with real-time access to the K8s Application Programming Interface (API) for performing AS actions. This paper specifically examines the integration of the DS NN architecture with popular RL algorithms to determine whether the learned policies can generalize effectively across various workload datasets and adapt to new, unseen workloads, thereby providing a robust solution for dynamic resource management. Fig. 1 illustrates our methodology that involves RL training on a given application (RC or OB) and directly applying the learned policy to another application without re-training due to the inherent capacities of the DS NN architecture since it is designed to process sets as inputs, addressing the unique challenges associated with the unordered and variable-sized nature of sets. Traditional NNs are typically not well-suited for sets because they expect inputs to be in a fixed order and of fixed size. DS overcome these limitations by using a permutation-invariant structure, meaning the network's output does not depend on the order of the elements in the input set. This capability is particularly valuable for addressing the dynamic nature of AS in modern cloud environments, where microservice instances may be frequently added to or removed from applications.

Deployment Information in *gym-hpa* is derived from K8s deployments to provide granular control over the resources allocated to each application a and its associated microservices m. The minimum ($\alpha_{m,\text{min}}$) and maximum ($\alpha_{m,\text{max}}$) replication

[1]https://github.com/jpedro1992/gym-hpa

[2]Example of collected datasets: https://zenodo.org/records/7944661

factors indicate the lower and upper bounds for the number of pod replicas that can be deployed for each microservice m. The request vector ($\gamma_{d,[r]}$) specifies the baseline resource allocation necessary for the stable operation of each microservice m, ensuring that the minimum required resources are always available. In contrast, the limit vector ($\Gamma_{d,[r]}$) sets the maximum permissible resource usage, preventing any single microservice from consuming an excessive amount of resources, which could lead to performance degradation or resource contention within the cluster [23]. Also, the total resource usage ($\rho_{m,[r]}$) is monitored, providing real-time data on the actual consumption of CPU and memory by each microservice m. Regarding the application's latency (Ψ_a), researchers can specify the particular measurement or metric to consider. The latency threshold (τ_a) for an application a sets the maximum acceptable response time, ensuring the application meets its performance requirements.

The observation space consists of five metrics per microservice deployment, namely the number of deployed pods, the CPU and memory consumption of the microservice, the percentage of CPU usage related to the CPU requests of the microservice, and the latency of the microservice. This information can be retrieved from the K8s cluster via its API or from the collected dataset if the simulation mode is enabled [12], [13]. For all these metrics, *min-max* normalization is applied in the observation space, a common practice in RL that helps the algorithm to converge faster and more reliably. It stabilizes the training for large input values, preventing the explosion of gradients.

The action space encompasses all possible actions an agent can select within the environment. The action space is *MultiDiscrete* [24], indicating a list of possible actions per discrete set. However, the agent can select only one action per discrete set at each step. The *Microservice* discrete set corresponds to the microservice selection, while the *Scaling* set consists of all existing scaling actions. The size of the action space depends on the total number of microservices in the application and the specified maximum and minimum replication factors. For example, if the maximum replication factor is four and the minimum is one, the maximum number of additions or terminations for each microservice is three. Therefore, the RL agent can choose the following actions:

- Keep the deployment running as is (*DoNothing*)
- Deploy additional pods (*Add*)
- Terminate a specified number of pods (*Stop*)

A Multi-Objective (MO) reward function (1) has been designed to incorporate three distinct objectives: **cost-aware**, **latency-aware**, and **resource-aware**. When the agent selects a valid action, it receives a positive reward based on these objectives, with corresponding weights (ω_c, ω_l, and ω_r). The total reward is normalized within the range of $[0.0, 1.0]$. Invalid actions, such as attempting to deploy or terminate pod instances that violate the maximum or minimum replication factors, result in a penalty of -1. This penalty encourages the agent to learn which actions are feasible given the current number of deployed pods, thereby refining its decision-making process.

$$\underbrace{r}_{Totalreward} = \underbrace{\omega_c \times r_c}_{Cost} + \underbrace{\omega_l \times r_l}_{Latency} + \underbrace{\omega_r \times r_r}_{Resources} \tag{1}$$

The cost-aware reward is designed to guide the agent in minimizing the number of instances (pods) allocated for each microservice deployment, thereby reducing overall deployment costs. The current deployment cost is calculated as the total number of deployed pods across all microservices in the application. This cost is then normalized based on the predefined minimum and maximum replication factors allowed in the experiment. The cost-aware reward is defined as the following:

$$r_c = 1 - \frac{\sum_{m=1}^{M} (R_m - \alpha_{m,\text{min}})}{\sum_{m=1}^{M} (\alpha_{m,\text{max}} - \alpha_{m,\text{min}})} \tag{2}$$

where:

- R_m is the number of deployed pods for microservice m.
- $\alpha_{m,min}$ is the minimum replication factor for m.
- $\alpha_{m,max}$ is the maximum replication factor for m.

The latency-aware function guides the agent to find suitable allocation schemes that minimize the overall application latency. The function is designed to provide a higher reward for lower latency by normalizing the application latency (Ψ_a) between 0 and a specified threshold (τ_a). The normalized latency $\hat{\Psi}_a$ is defined as:

$$\hat{\Psi}_a = \min\left(\frac{\Psi_a}{\tau_a}, 1\right) \tag{3}$$

The latency-aware reward is then formulated as:

$$r_l = \begin{cases} 1 - \hat{\Psi}_a & \text{if } \Psi_a \leq \tau_a \\ 0 & \text{if } \Psi_a > \tau_a \end{cases} \tag{4}$$

The resource-aware reward aims to find allocation schemes that ensure efficient resource utilization while avoiding over-provisioning or under-provisioning schemes based on the percentage of CPU usage relative to the amount requested by each pod. First, the CPU utilization percentage for each microservice m is calculated using the following formula:

$$\Theta_{m,[cpu]} = \frac{\rho_{m,[cpu]}}{\gamma_{m,[cpu]}} \times 100 \tag{5}$$

where:

- $\rho_{m,[cpu]}$ is the CPU usage of microservice m.
- $\gamma_{m,[cpu]}$ corresponds to the total amount of CPU requested for microservice m.
- $\Theta_{m,[cpu]}$ is the CPU utilization percentage in terms of the amount of CPU requested for microservice m.

The percentage $\Theta_{m,[cpu]}$ is then normalized between 0 and 100 since it can be above 100% because resource requests are typically lower than resource limits in typical K8s deployments [23]. Thus, the normalized percentage $\hat{\Theta}_{m,[cpu]}$ for microservice m is defined as:

$$\hat{\Theta}_{m,[cpu]} = \min\left(\frac{\Theta_{m,[cpu]}}{100}, 1\right) \tag{6}$$

Finally, the resource-aware reward is then calculated by considering all microservices in the application:

$$r_r = 1 - \frac{1}{M}\sum_{m=1}^{M} \hat{\Theta}_{m,[cpu]} \tag{7}$$

Cloud administrators face the ongoing challenge of balancing different performance factors. For example, ensuring low deployment costs and high resource efficiency while minimizing latency in their cloud infrastructure. Their ultimate decisions are made based on user needs and organizational objectives. Although prioritizing costs can help minimize deployment expenses, particularly for budget-constrained companies, favoring low latency is crucial for real-time applications where responsiveness is key. This paper evaluates the trade-off of three distinct strategies while showcasing their benefits and disadvantages.

The DeepSets (DS) Neural Network (NN) [11], [25] is a NN architecture specifically designed to process sets as inputs, addressing the challenges associated with their unordered and variable-sized nature. Traditional NNs, which typically expect inputs to be ordered and of fixed size, are not well-suited for handling sets. However, DS overcomes these limitations by employing a permutation-invariant structure, ensuring that the network's output is independent of the order of elements in the input set. The architecture of DS is typically expressed as:

$$f(X) = \rho\left(\sum_{x\in X} \phi(x)\right), \tag{8}$$

where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is the input set.
- $\phi(x)$ is an embedding function applied to each element x in the set.
- The summation $\sum_{x\in X}$ is a permutation-invariant operation that aggregates the embeddings.
- $\rho(\cdot)$ is a function that processes the aggregated result to produce the final output.

As a result, in DS NNs, both inputs and outputs can be arbitrarily sized sets. This flexibility allows the RL algorithm to learn a policy that can be applied across different application scenarios with varying numbers of microservices. Thus, the main benefits of using DS in the context of dynamic AS include:

- **Permutation Invariance:** Since the order of elements in a set is inherently irrelevant, DS respect this property, ensuring that the output is the same regardless of how the input elements are arranged.
- **Variable-Sized Inputs:** In contrast to traditional NNs that require fixed-sized inputs, DS can process inputs of varying sizes, making them suitable for applications where the number of microservices may change dynamically.

TABLE I: The evaluated reward strategies.

Name	ω_c	ω_l	ω_r
Cost	1.0	0.0	0.0
Latency	0.0	1.0	0.0
Resources	0.0	0.0	1.0
FavorCost	0.5	0.25	0.25
FavorLat	0.25	0.5	0.25

TABLE II: Deployment properties of both applications.

App.	Deployment	CPU R/L (in m)	RAM R/L (in Mi)	Rep. (α_m)
RC (a_1)	*Leader & Follower*	250/500	250/500	1/8
OB (a_2)	*Frontend & Recomm.*	100/200	64/128	1/8
	Cart & Add	200/300	180/300	
	Product & Currency	100/200	64/128	
	Payment & Shipping	100/200	64/128	
	Email & Checkout	100/200	64/128	
	Redis-cart	70/125	200/256	

- **Scalability:** DS can easily scale to handle larger sets by simply aggregating more elements. This is particularly useful in cloud environments, where the number of microservices can be high.
- **Generalization:** The architecture allows RL algorithms to learn policies that generalize across different workload scenarios. For instance, a policy trained on one application can be applied to another without retraining, thanks to the ability of DS to handle diverse sets of inputs.

IV. Evaluation Setup

Reward Strategies Table I details five distinct reward strategies considered in the evaluation. By evaluating numerous reward strategies, our aim is to provide insights into the effectiveness of different auto-scaling strategies for microservice-based applications in containerized environments. **RL Algorithm** A notable algorithm known as Proximal Policy Optimization (PPO), which supports *MultiDiscrete* action spaces, has been evaluated using the stable baselines 3 and CleanRL [26], [27] libraries. This implementation has been adapted to utilize the DS NN architecture by modifying the standard implementations from these popular RL libraries. By utilizing the DS NN, our objective is to assess the feasibility of cross-dataset evaluation, where the RL algorithm is trained on one microservice application and tested on a completely different one.

Applications Table II shows the deployment properties for the evaluated applications. Different resource requests and limits (i.e., CPU and memory) have been specified for each microservice. The minimum and maximum replication factors are set to 1 and 8, respectively. The first application relates to the deployment of the RC microservice [14] consisting of two K8s deployments: *Leader* and *Follower*. RC is a highly-available application, so AS mechanisms should ensure no downtime during scaling operations. The latency for RC (Ψ_{a_1}) corresponds to the calculation of the average response time of the Redis server by collecting the total duration of the query and the total response time of processing the query during the last five minutes, as shown in (9). The latency threshold (τ_{a_1}) is set to 250 milliseconds. The second scenario relates

TABLE III: The execution time during RL training.

Strategy	App.	Execution Time per episode (in s)	Execution Time for 8000 episodes
Cost	RC	0.35 ± 0.01	46.6 min (0.77 hours)
	OB	1.56 ± 0.02	208.0 min (3.46 hours)

to the OB application [15] consisting of 11 K8s deployments. It is a web-based e-commerce application where users can browse items and add them to their cart to purchase them. The *Frontend* service receives HTTP requests and forwards them to several services, including *Currency* and *Product*. The latency for OB (Ψ_{a_2}) corresponds to the average response time based on the *GET /cart* request as shown in (10), measured using the Locust load testing tool [28]. This request represents a critical user interaction point within the OB's functionality. Although we also explored the average response time of various requests, our experiments revealed similar results. Thus, we chose to focus on the response time of the *GET /cart* request, as it provides a suitable representation of user experience while reducing the number of API calls to Prometheus. The latency threshold (τ_{a_2}) is set to 3 seconds.

$$\Psi_{a_1} = \frac{redis\_commands\_duration\_sec\_total[5m]}{redis\_commands\_proc\_total[5m]} \quad (9)$$

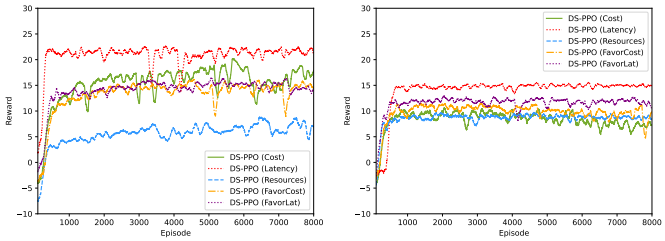
$$\Psi_{a_2} = locust\_avg\_response\_time\_GET\_cart \quad (10)$$

The *gym-hpa* Framework has been used in the evaluation. An episode consists of 25 steps during which the RL agent aims to maximize the reward based on the current demand and the number of pods deployed for each microservice in the application. During the experiments, the RL agents have been trained over 8000 episodes (200K steps) and then tested for 100 episodes, utilizing a 14-core Intel i7-12700H CPU @ 4.7 GHz processor with 16 GB of memory. The performance of the RL agents has been evaluated based on the following metrics:

- **Accumulated reward** during each episode. It refers to the total sum of rewards obtained by an RL agent throughout each episode.
- **Average number of deployed pods** during an episode.
- **Average latency (in ms)** for the application during an episode.
- **Average CPU percentage** related to the pod's CPU request amount (in m).

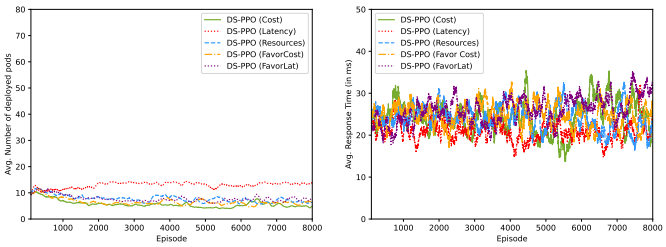
V. Results

Execution Time Table III presents the execution times for training using the DS-PPO algorithm for both applications. The results reveal a significant difference in the computational demands between the two applications. For the RC application, the execution time per episode is relatively low, with the *Cost* strategy needing 0.35 seconds per episode on average, resulting in a total training time of approximately 46.6 minutes for 8000 episodes. In contrast, the OB application exhibits a much higher execution time per episode on average. The



(a) Redis Cluster (RC). (b) Online Boutique (OB).

Fig. 2: The accumulated reward during training.



(a) RC - Number of Pods. (b) RC - Latency.

Fig. 3: Results obtained during training for the RC application.

Cost strategy requires an average of 1.56 seconds per episode, leading to a total training time of around 3.46 hours for 8000 episodes. The longer execution time for the OB application highlights its greater complexity, as the environment includes eleven microservices, resulting in a significantly larger observation space.

Training Phase Fig. 2 illustrates the accumulated reward for both applications across different strategies. The DS-PPO algorithm achieves slightly higher rewards for the RC application than the OB. In both scenarios, the *Latency* strategy achieved the highest accumulated rewards. In addition, Fig. 3 showcases the performance of DS-PPO for the RC application in terms of the number of deployed pods, and the application latency. For the RC application, the *Cost* strategy effectively minimizes deployment costs by deploying fewer pods. This approach is ideal for scenarios where cloud administrators aim to minimize operational costs as their primary concern. On the other hand, the *Latency* strategy prioritizes performance by deploying more pods on average, resulting in slightly lower application latency. This strategy is well-suited for latency-sensitive environments where performance is critical, though it generally incurs higher deployment costs. Furthermore, the *FavorCost* and *FavorLat* strategies successfully guide the agent toward a balanced policy, aligning with the intended goals of the MO reward function. These strategies achieve a compromise between deployment costs, resource efficiency, and application performance, making them versatile options for environments where a balance between these performance factors is desired. For the OB application, differences between strategies are less pronounced due to lower rewards and the inherent complexity of its 11 microservices. To address this, we leverage the versatility of DS to transfer the policy trained in RC directly to OB without retraining.

TABLE IV: Results obtained during the testing phase for both applications, including cross-dataset experiments. The results illustrate the impact of strategy selection on key performance factors and the challenges in RL generalization across different applications. Also, compared to our prior work [12], DS-PPO consistently achieves higher performance for both applications.

Algorithm	Training	Testing	Strategy	Accumulated Reward	Number of Pods	Latency (in ms)	CPU requests (in %)
DS-PPO	RC	RC	*Cost*	22.07 ± 0.33	**3.56 ± 0.17**	33.22 ± 10.04	24.45 ± 2.27
			Latency	**24.56 ± 0.39**	7.40 ± 0.49	**4.37 ± 3.86**	28.92 ± 1.99
			Resources	12.01 ± 1.0	4.44 ± 0.19	8.12 ± 7.03	**32.08 ± 2.19**
			FavorCost	18.20 ± 0.47	5.71 ± 0.44	**2.71 ± 2.30**	29.53 ± 1.96
			FavorLat	19.50 ± 0.40	6.84 ± 0.39	**2.80 ± 2.09**	29.12 ± 2.01
DS-PPO	RC	OB	*Cost*	**16.98 ± 0.46**	**35.70 ± 1.42**	1030.43 ± 9.96	81.01 ± 2.50
			Latency	16.29 ± 0.14	46.61 ± 1.78	1044.94 ± 17.0	62.82 ± 3.74
			Resources	11.94 ± 0.32	45.87 ± 1.60	1038.05 ± 12.89	66.88 ± 3.87
			FavorCost	12.70 ± 1.06	47.64 ± 1.90	1036.57 ± 13.31	63.04 ± 4.15
			FavorLat	14.42 ± 0.23	46.61 ± 1.78	1044.94 ± 17.0	62.82 ± 3.74
A2C [12]	RC	RC	*Cost-aware*	-	4.50 ± 0.4	25.97 ± 2.67	12.36 ± 0.57
			Latency-aware	-	12.81 ± 1.19	18.84 ± 0.59	6.4 ± 0.6
A2C [12]	OB	OB	*Cost-aware*	-	54.03 ± 3.96	1034.77 ± 4.4	33.40 ± 2.04
			Latency-aware	-	63.79 ± 4.05	1025.64 ± 20.24	33.34 ± 2.09

Testing Phase Given the increased complexity and challenges associated with the training of the OB use case, the testing phase focuses exclusively on models trained using the RC application. Table IV provides detailed results from the testing phase, evaluating the DS-PPO algorithm's performance across various strategies on both the RC and OB applications, including cross-dataset experiments. When testing on the same dataset, the *Latency* strategy achieved the highest accumulated reward, while achieving lower latency than other strategies. However, it required more deployed pods (7.4) and slightly higher CPU requests (28.92%). The *Cost* strategy showed the potential towards minimizing the deployment cost since it required fewer pods (3.56) and lower CPU requests (24.45%).

The cross-dataset testing from RC to OB emphasizes the versatility and adaptability of the DS NN architecture. The fact that the trained models in RC delivered acceptable performance when tested in OB without requiring retraining, highlights its robustness in a more challenging use case. Although the performance of the model experienced some degradation as expected, and most of the strategies yielded lower rewards, the *Cost* strategy achieved the highest accumulated reward of 16.98 while maintaining the lowest number of pods at 35.70. Other strategies faced difficulties in further optimizing the allocation scheme for the OB application. For example, latency values remained relatively consistent across strategies. These numbers reflect the challenges of applying trained RL models to different applications, especially when transitioning to a significantly more complex application as OB, which consists of 11 microservices compared to the 2 microservices in RC. The fact that trained strategies could still deliver acceptable performance under these demanding conditions highlights the generalization capabilities and robustness provided by DS. In summary, these results illustrate the challenges associated with RL model generalization.

Comparison with previous work [12] The lower section of Table IV presents the performance of the Advantage Actor Critic (A2C) algorithm, identified as one of the top-performing algorithms in [12], across single-reward objectives: *Cost-aware* and *Latency-aware*. DS-PPO consistently outperforms A2C for both applications, particularly when it comes to minimizing cost or reducing latency. For instance, for the RC application, DS-PPO maintains a smaller average number of pods compared to A2C under the *Cost-aware* strategy. Also, DS-PPO achieves lower latency than A2C for the latency-aware strategy. A2C performs well for the OB application when using the *Latency-aware* strategy, as it effectively reduces latency. However, this comes at the cost of a significantly higher number of pods compared to DS-PPO. This comparison highlights the advantages of MO reward strategies and advanced policy optimization methods such as DS-PPO, which can help achieve more adaptable resource management policies.

VI. Conclusions

This paper explored the application of RL for AS in containerized cloud environments. The main research purpose has been to assess whether RL approaches could learn adaptive scaling policies that generalize across diverse workloads and perform effectively on unseen workloads. A MO reward function has been designed focusing on key performance factors such as the application's response time, deployment cost, and resource utilization. The proposed DS-PPO algorithm achieved adequate performance for the OB application when trained using data from the RC application. The Cost strategy achieved the highest accumulated reward of 16.98 while maintaining the lowest number of pods at 35.70. Results show that RL effectively balances competing objectives and adapts to changing workloads. This adaptability is key in dynamic cloud environments with unpredictable workloads. Future work will focus on refining the MO reward function, incorporating additional performance metrics, and exploring multi-agent RL for AS, where each RL agent manages a single microservice. RL presents a promising approach for improving AS in containerized cloud environments, offering more adaptive and efficient strategies than traditional methods.

Acknowledgment

José Santos is funded by the Research Foundation Flanders (FWO), grant number 1299323N.

REFERENCES

[1] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2557–2589, 2021.

[2] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th international middleware conference*, 2016, pp. 1–13.

[3] J. Dogani, R. Namvar, and F. Khunjush, "Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey," *Computer Communications*, 2023.

[4] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.

[5] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "Delivering elastic containerized cloud applications to enable devops," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2017, pp. 65–75.

[6] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *2017 IEEE 10th international conference on cloud computing (CLOUD)*. IEEE, 2017, pp. 472–479.

[7] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 33–40.

[8] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Fog computing: Enabling the management and orchestration of smart city applications in 5g networks," *Entropy*, vol. 20, no. 1, p. 4, 2017.

[9] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE communications surveys & tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[10] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE communications surveys & tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[11] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola, "Deep sets," *Advances in neural information processing systems*, vol. 30, 2017.

[12] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "gym-hpa: Efficient auto-scaling via reinforcement learning for complex microservice-based applications in kubernetes," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2023, pp. 1–9.

[13] J. Santos, E. Stratos, T. Wauters, B. Volckaert, and F. De Turck, "*Gwydion*: Efficient auto-scaling for complex containerized applications in kubernetes through reinforcement learning," in *Under Review in Journal of Network and Computer Applications*. Elsevier, 2024.

[14] Redis, "Redis, an open source in-memory data structure store." accessed on 6 August 2024. [Online]. Available: https://redis.io/.

[15] O. Boutique, "Online boutique, a cloud-native microservices demo application." accessed on 6 August 2024. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo.

[16] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*. IEEE, 2017, pp. 64–73.

[17] S. Rahman, T. Ahmed, M. Huynh, M. Tornatore, and B. Mukherjee, "Auto-scaling vnfs using machine learning to improve qos and reduce cost," in *2018 IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–6.

[18] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 329–338.

[19] Y. Wei, D. Kudenko, S. Liu, L. Pan, L. Wu, and X. Meng, "A reinforcement learning based auto-scaling approach for saas providers in dynamic cloud environment," *Mathematical Problems in Engineering*, vol. 2019, no. 1, p. 5080647, 2019.

[20] W. Iqbal, A. Erradi, M. Abdullah, and A. Mahmood, "Predictive auto-scaling of multi-tier applications using performance varying cloud resources," *IEEE Transactions on Cloud Computing*, vol. 10, no. 1, pp. 595–607, 2019.

[21] D. Lee, J.-H. Yoo, and J. W.-K. Hong, "Deep q-networks based auto-scaling for service function chaining," in *2020 16th International Conference on Network and Service Management (CNSM)*. IEEE, 2020, pp. 1–9.

[22] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[23] Kubernetes, "Resource management for pods and containers." accessed on 6 August 2024. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.

[24] openAIGym, "The action spaces in openai gym." accessed on 6 August 2024. [Online]. Available: https://github.com/openai/gym/tree/master/gym/spaces.

[25] N. Di Cicco, G. F. Pittalà, G. Davoli, D. Borsatti, W. Cerroni, C. Raffaelli, and M. Tornatore, "Drl-forch: A scalable deep reinforcement learning-based fog computing orchestrator," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. IEEE, 2023, pp. 125–133.

[26] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[27] S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. Araújo, "Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms," *Journal of Machine Learning Research*, vol. 23, no. 274, pp. 1–18, 2022. [Online]. Available: http://jmlr.org/papers/v23/21-1342.html

[28] Locust, "An open source load testing tool." accessed on 6 August 2024. [Online]. Available: https://locust.io/.