Towards Optimal Load Balancing in Multi-Zone Kubernetes Clusters via Reinforcement Learning

José Santos*, Tim Wauters*, Filip De Turck*, Peter Steenkiste†
* Ghent University - imec, IDLab, Department of Information Technology, Gent, Belgium
Email: {josepedro.pereiradossantos, tim.wauters, filip.deturck}@UGent.be
† Department of Computer Science and Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA
Email: {prs}@cs.cmu.edu

***Abstract*—With the advent of container technology, companies have been developing microservice-based applications, converting the old monolithic software into a group of loosely coupled containers, with the aim of offering greater flexibility and improving operational efficiency. When users access microservices, their initial point of contact is typically a load balancer. This component is responsible for distributing incoming traffic or requests between multiple instances of microservices. Traditional load balancing approaches mainly rely on round-robin, or weighted round-robin algorithms which are inadequate to maintain the overall performance and scalability of microservice-based applications. Microservices are often deployed in dynamic environments needing a more adaptive and efficient load balancing strategy to optimize resources and reduce the overall latency for end users. This paper presents a dynamic load balancer for Kubernetes (K8s) clusters based on Reinforcement Learning (RL). It aims to minimize the overall latency while promoting fair distribution of requests. To achieve this goal, the load balancer considers both current network delays and processing loads in the cluster. The evaluation shows that our solution is effective even in environments where both the network traffic and the processing loads in the cluster change dynamically over time. In addition, this study highlights the flexibility of DeepSets neural networks in solving the load balancing challenge in diverse setups without retraining. The results show that the DeepSets algorithms can solve the microservice load balancing problem even in scenarios up to 30 times larger than the trained setup.**

***Index Terms*—Load Balancing, Reinforcement Learning, Microservices, Kubernetes, Containers**

I. Introduction

In recent years, software development has witnessed a significant shift towards microservices [1], [2]. Microservice-based architectures involve decomposing traditional monolithic applications into several loosely coupled containers that aim to enhance flexibility and operational efficiency. Users typically access these microservices via a *load-balancer* that is responsible for distributing incoming traffic or requests between multiple replicas of a given microservice. Conventional load balancing algorithms in popular container orchestration platforms such as Kubernetes (K8s) [3] aim to maintain a scalable, resilient, and high-performance architecture. However, traditional mechanisms are insufficient to sustain the performance and scalability demanded by microservices-based applications [4].

Most load balancing algorithms [5] favor round-robin strategies, focusing on distributing the load evenly among available microservice instances. These approaches ignore performance properties of the underlying cloud infrastructure and the real-time resource consumption in the cluster. This raises several research questions: *Which microservice instance to choose to handle a request without compromising performance? Can the selected instance provide low end-to-end latency to the user? Should the load balancing algorithm focus on reducing the overall latency or promoting equal distribution of requests among deployed microservices?*

Network latency poses a significant challenge in microservice-based architectures, impacting the user experience and application performance [6]. With microservices distributed across various nodes within a multi-zone K8s cluster, efficient load balancing mechanisms are crucial for maintaining low-latency communications and evenly distributing the load over the available microservice replicas. While suitable for distributing traffic evenly, traditional load balancing methods often fail to prioritize latency reduction, resulting in suboptimal user experiences. As users increasingly demand seamless and responsive applications, addressing these latency issues becomes essential for ensuring the wide adoption of microservice-based architectures in future cloud infrastructures [7], [8]. Also, in dynamic cloud environments, microservice instances are constantly being added or terminated based on the current demand, which increases the need for novel intelligent load balancing mechanisms. Modern load balancing approaches need to consider the current resource consumption of microservices (e.g., CPU and memory usage) but also anticipate and adapt to fluctuations in network traffic to reduce the overall latency (i.e., processing and network latency) and optimize system performance.

This paper tackles these challenges by studying a Reinforcement Learning (RL)-based approach for microservice load balancing within a multi-zone K8s cluster. An RL environment named *gym-loadbalancing* has been developed to provide a scalable and cost-effective solution to train RL agents to address this problem. Our results show that RL can find near-optimal load balancing schemes, prioritizing overall latency reduction and avoiding distribution inequality compared against heuristic-based methods. In addition, this work assesses the ability of DeepSets neural networks [9] to generate models that address the load balancing challenge across a range of

conditions without needing retraining. Inputs and outputs are arbitrarily-sized sets in the DeepSets neural network, meaning that the policy learned by the RL algorithm can be applied to diverse multi-zone scenarios with a varying number of available endpoints. The main contributions of the paper are threefold:

- ***gym-loadbalancing* framework**: Implementation of an RL-based framework for microservice load balancing in multi-zone K8s clusters. The proposed framework[1] has been open-sourced, allowing researchers to evaluate their algorithmic ideas. Sec. IV presents the RL approach, including details on the observation state, action space, and the considered multi-objective reward function. The approach focuses on reducing the overall latency and promoting equal distribution of requests.
- **Extensive Evaluation**: Our experiments considered different RL algorithms and several heuristic-based methods applied in multiple multi-zone K8s scenarios under dynamic conditions, where processing and network latency vary as well as the CPU usage of microservices (Sec. V). Results show that RL can find appropriate load balancing schemes for the selected strategy while achieving higher performance than typical heuristics.
- **RL generalization**: This work also evaluates the generalization potential of the DeepSets neural network by applying it to different problem sizes without retraining. Results show that DeepSets algorithms can optimize microservice load balancing in different multi-zone scenarios 30 times higher than the trained scenario (Sec. VI).

The paper is organized as follows: the state-of-the-art on microservice load balancing is discussed in the next section. Sec. III highlights the importance of efficient microservice load balancing, describing how load balancing is currently handled within the K8s platform. Sec. IV details the RL approach, including observation and action spaces. Sec. V describes the evaluation setup, followed by the results in Sec. VI. Finally, Sec. VII summarizes our results.

II. Related Work

This section reviews the most relevant works on microservice load balancing, mainly focusing on algorithms for multi-zone K8s infrastructures. The awareness of the load balancer plays a major role in these scenarios given the differences in network bandwidth and latency inside and between zones.

Heuristics and Theoretical Formulations have been widely studied in the literature [4], [10]–[14]. For example, in [4], the authors propose a chain-oriented load balancing algorithm based on message queues. The goal is to efficiently distribute the load among microservices to minimize their response time. Also, in [11], Yu, R. et al. introduce a graph-based model to represent load dependencies among microservices. While their formulation can be solved optimally in polynomial time, it does not explicitly consider the Quality of Service (QoS) of the application. Furthermore, in [12], the authors present a chain-based load balancing algorithm focused on the system resource usage of each service instance. The study evaluates three heuristics, demonstrating the efficacy of the proposed method in reducing response time when compared to existing approaches. One notable limitation of these heuristics lies in their platform-specific design, restricting their broader practical applicability. Moreover, theoretical models often exhibit slow convergence and are typically challenging to implement in production environments.

Load Balancing in K8s is an active research topic [15]–[17]. Many proposals focus on enhancing the load balancing mechanisms within K8s through monitoring information [15], container migration [16], or leader-election algorithms [17]. Nonetheless, most works still do not address the underlying cluster infrastructure topology nor do they provide a comprehensive study on the impact of distribution inequality in the load balancing system. Single-zone clusters are typically evaluated using performance metrics such as response time and resource consumption of the application without considering load distribution inequality or network latency.

This study introduces a novel approach based on RL, aiming to make efficient load balancing considering both the current status of the multi-zone K8s infrastructure and the microservice applications. We use a multi-objective reward function that focuses on reducing the overall latency for end users while fostering an equal distribution of requests across the available microservice instances. To the best of our knowledge, RL techniques have not been previously proposed to handle the microservice load balancing problem discussed in this paper, and most works do not consider all the performance factors evaluated in this work. RL approaches are generally robust to dynamic demands since the algorithm adjusts the model parameters if any notable event occurs (i.e., online learning). However, a key drawback of RL techniques is their high execution time to converge to a stable model, potentially leading to inefficient actions during the learning period. To address this limitation, the proposed approach introduces a more scalable and cost-effective solution for RL training through the use of the *gym-loadbalancing* framework, a near-real simulation-based environment.

III. Microservice Load Balancing in Kubernetes

K8s offers robust solutions for microservice load balancing, providing dynamic and adaptive mechanisms to distribute incoming traffic across microservice instances [18]. K8s introduces the concept of a *Service* as an abstraction that defines a logical set of *Pods*, the smallest working unit in K8s that can host one or more containers, as shown in Fig.1. The *Service* abstraction provides a *Service Load Balancer* that balances incoming traffic across the associated *Pods*. This abstraction simplifies the process of load balancing for microservices, by hiding unnecessary complexity. Nevertheless, traditional load balancing approaches struggle to keep pace with the dynamic changes inherent in microservices architectures. Individual microservices often exhibit varying workloads based on user demand and application complexity. Efficient load balancing

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

mechanisms are crucial to prevent overloading of specific instances and ensure computing resources are used efficiently. Service meshes, such as Istio [19] and Linkerd [20], enhance K8s load balancing by providing advanced traffic management features, including retries and traffic shifting. These features contribute towards a higher resilience and fault tolerance of microservices in K8s clusters.

Load balancing in *multi-zone* K8s clusters is even more challenging due to the inherent geographical distribution. The communication between zones (i.e., inter-zone communication) incurs additional overhead compared to intra-zone communication. The proposed load balancing approach uses a higher network cost between zones in the K8s cluster as outlined in [21]. Our previous work introduced the concept of a NetworkTopology Custom Resource Definition (CRD) [2] to save network costs between regions and zones for the underlying K8s cluster topology. In addition, the integration of a netperf component [3] allows for the estimation of network latency between cluster nodes at various percentiles (i.e., 50th, 90th and 99th percentile). These measurements can serve as network costs, aligning with the expected network latency within the cluster. In the context of the proposed RL approach, these network costs play a pivotal role in optimizing request routing to minimize inter-zone communication. Also, the heterogeneous computing capacities across zones necessitate load balancing strategies that dynamically adapt to resource variations (e.g., CPU usage of microservices). Balancing the load accordingly to the resource heterogeneity ensures optimal resource utilization and prevents overloading specific zones.

End-to-end system design In our OpenAI Gym-based RL load balancing system [22] named as *gym-loadbalancing*, several RL algorithms are available for training to generate a load balancing policy using as input static and dynamic information about the K8s cluster (detailed further in Sec. IV). For example, load information varies dynamically for both the intra-zone network latency (assuming netperf is periodically executed and measurements are available in the cluster) and the microservices CPU usage and processing latency based on the total number of requests assigned to the endpoint. This information serves as input and is updated periodically after each action selected by the RL algorithm. The *gym-loadbalancing* framework has been developed to replicate the behavior of load balancing requests in a K8s cluster, providing the RL agent with relevant information available within a typical K8s cluster. By emulating the behavior of a K8s cluster, trained RL load balancing policies could then be later validated in operational environments by retrieving real-time information from the K8s cluster, such as our previously developed components (e.g., NetworkTopology CRD and netperf), and popular monitoring platforms such as Prometheus [23]. Though, this validation is left out of the scope of this paper, but planned as future work since the K8s sig-network community is currently testing and validating the implementation of a pluggable framework [4] that will allow researchers and developers to implement their load balancing algorithms as plugins and use them in typical K8s clusters. Since this implementation is still under review, our RL approach is validated in the *gym-loadbalancing* framework, described in detail in the next section.

[2]https://github.com/diktyo-io/networktopology-api

[3]https://github.com/jpedro1992/pushing-netperf-metrics-to-prometheus

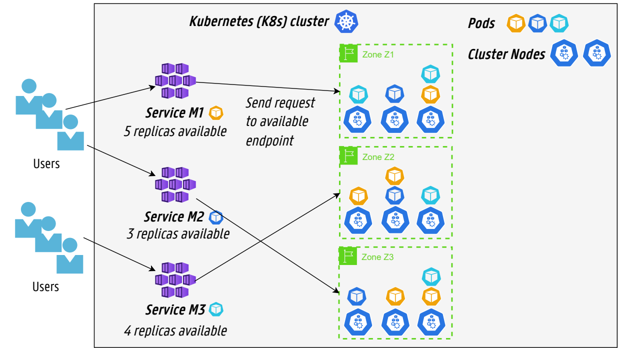


Fig. 1: Illustration of microservice load balancing in K8s.

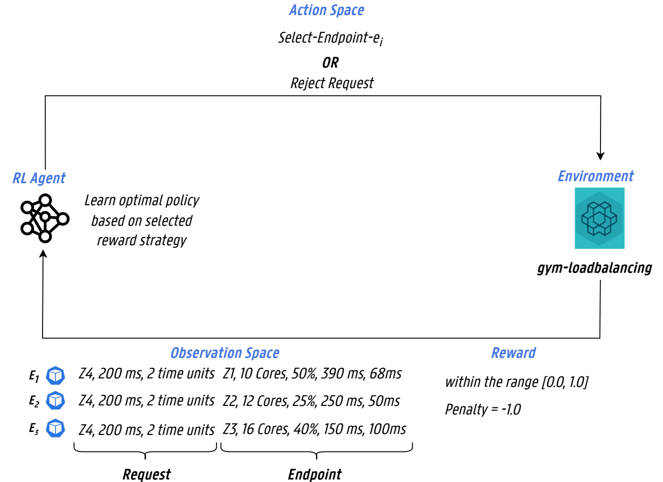


Fig. 2: Overview of the *gym-loadbalancing* framework.

IV. Reinforcement Learning for Microservice Load Balancing in Kubernetes

The gym-loadbalancing framework has been developed to train RL algorithms in a scalable and cost-effective manner as illustrated in Fig. 2. These OpenAI Gym-based environments are developed to speed up the training process of RL. During training, dynamic load information (e.g., number of available endpoints and its corresponding metrics) is updated based on the chosen actions of the RL agent in the *gym-loadbalancing* environment. Sec. V shows the deployment requirements used for the RL environment based on a realistic microservice-based application to create near-real experiments.

In addition, the proposed approach adopts the DeepSets methodology presented in [24], [25]. Deep RL methods based on Multi-Layer Perceptrons (MLPs) operate in fixed-length vector spaces, which cannot support variable input or output dimensionalities. In other words, for the microservice load

[4]https://github.com/kubernetes/kubernetes/pull/121060

TABLE I: The structure of the Observation Space.

Set	Metric	Description
Request	Z_{in}	The zone of the endpoint.
	Δ_r	The latency threshold of the request r.
	T_r	The expected duration of the request r.
Endpoint	Z_{out}	The zone of the endpoint e.
	Π_{out}	The zone cpu capacity.
	Θ_e	The current CPU usage of the endpoint e.
	ϕ_e	The endpoint e processing latency (in ms).
	δ_e	The network latency from Z_{out} to Z_{in} (in ms).

TABLE II: The hardware configuration of each node based on Amazon EC2 On-Demand Pricing [26].

Node Type	Amazon Image	CPU capacity	Memory capacity
Cloud	t4g.2xlarge	8.0	32.0
Fog Tier 2	t4g.xlarge	4.0	16.0
Fog Tier 1	t4g.large	2.0	8.0
Edge Tier 2	t4g.medium	2.0	4.0
Edge Tier 1	t4g.small	2.0	2.0

balancing problem, if an MLP-based RL agent trains on a multi-zone setup with four endpoints, it cannot be directly applied to another multi-zone scenario that consists of eight endpoints. Instead, the DeepSets neural network assumes that inputs and outputs can be arbitrarily-sized sets, so the learned policy by the RL algorithm is not bound to a fixed number of endpoints. Thus, a DeepSets-based RL agent can generalize its learned policy to different multi-zone scenarios with a varying number of available endpoints without retraining. By applying DeepSets, this work aims to find RL algorithms that generalize well to problem sizes larger than training, which is beneficial since microservice replicas are frequently scaled up or down in response to dynamic demands, meaning that the number of available endpoints varies significantly in typical K8s clusters. More generally, changes in the number of endpoints do not require retraining. Retraining is only needed when new metrics are added to the observation space.

Observation Space Table I shows the observation space considered for the microservice load balancing problem, describing the environment at a given step. It includes two sets of metrics: *Request* and *Endpoint*. The first set *Request* corresponds to the metrics related to the endpoint generating the request, such as the zone hosting the endpoint (Z_{in}), the latency threshold (Δ_r) of the load balancing request that the RL agent should respect, and the expected duration (T_r) of the request r measured in time units.

The second set *Endpoint* corresponds to the metrics related to the current status of all available endpoints (i.e., microservice replicas), such as the hosting zone (Z_{out}), the zone CPU capacity (Π_{out}), the current CPU usage of the endpoint (Θ_e), the processing latency of the endpoint (ϕ_e), and the network latency of the endpoint determined by the cluster topology (δ_e). The CPU usage and processing latency of endpoint e are directly influenced by the current number of requests being handled by this endpoint. As the number of requests increases, both the CPU usage and processing latency of the endpoint e increase. The network latency and the endpoint's processing latency are quantified within the range of $[1.0, 500.0]$ milliseconds. Inter-zone communication

TABLE III: The structure of the Action Space.

Action Name	Description
Select-e	The endpoint e is selected to handle the request.
Reject	The agent rejects the request.

varies dynamically during the experiments within the range of $[1.0, 500.0]$ milliseconds, while intra-zone communication is assumed as 1.0 ms. Table II shows detailed resource capacities for each node based on different cluster types. The zone CPU capacity is derived from these capacities, expressed as values within the interval $[2.0, 32.0]$ based on the total number of nodes available per zone. This metric is mainly used by the Zone-Greedy heuristic approach presented in Sec. V.

Action Space Table III shows the action space designed for *gym-loadbalancing* as a discrete set of possible actions, where a single action is chosen at each step. Given a request, the RL agent decides to forward the request to a given endpoint, or reject the request. The size of the action space depends on the total number of endpoints in the scenario. Let us assume that the setup consists of e endpoints, the action space length is then $e + 1$. Rejection is allowed since the agent might prefer to reject a request since its endpoints might be serving several requests, though the agent is penalized in these cases. Regarding penalties (i.e., negative reward), a simple approach commonly followed in the literature [27] is to penalize the agent if it selects an invalid action since these are typically known beforehand. This penalty should align with the reward function to ensure that the agent can achieve the predetermined goal. This work focuses on a multi-objective reward function, outlining two opposing strategies, as described next.

Reward Functions The purpose of a reward function is to guide the RL agent towards maximization of accumulated rewards by choosing appropriate actions depending on the observation state. Our multi-objective reward function (1) considers two complementary objectives: latency-aware (2), and load inequality-aware (3). If the agent chooses an endpoint, it receives a positive reward based on both strategies and its corresponding weights (ω_l and ω_i), normalized between $[0.0, 1.0]$, otherwise, the agent is penalized if it decides to reject the request (i.e., -1).

$$r = \begin{cases} \omega_l \times r_{latency} + \omega_i \times r_{inequality} & \text{if req. is accepted.} \\ -1 & \text{if req. is rejected.} \end{cases} \tag{1}$$

$$r_{latency} = 1.0 - \lambda_r \quad \text{where:} \quad \underbrace{\lambda_r}_{\text{Total latency}} = \underbrace{\phi_e}_{\text{Processing}} + \underbrace{\delta_e}_{\text{Network}} \tag{2}$$

$$r_{inequality} = 1.0 - G \quad \text{where:} \quad G = \textit{Gini Coefficient} \tag{3}$$

On the one hand, the latency-aware function aims to minimize the total latency of the request (λ_r) by reducing both the endpoint processing (ϕ_e) and network latency (δ_e). On the other hand, the inequality-aware function leads the RL agent

TABLE IV: The evaluated reward strategies.

Name	ω_l	ω_i
Latency	1.0	0.0
Inequality	0.0	1.0
Multi-goal	0.5	0.5

TABLE V: Load Balancing requirements of TeaStore.

Microservice	Latency Threshold (Δ)
webui	400 ms
registry	200 ms
image	150 ms
auth	250 ms
persistence	450 ms
db	375 ms
recommender	500 ms

to choose endpoints that evenly distribute requests across the number of available endpoints. The reward is calculated based on the *Gini Coefficient* (G) that ranges from $[0.0, 1.0]$, where 0 means perfect equality (all endpoints serve the exact number of requests), and 1 indicates perfect inequality (one endpoint serves all the requests). A lower *Gini Coefficient* indicates then a more equitable distribution. The *Gini Coefficient* is an accurate measure of inequality in a distribution, calculated using the formula:

$$G = \frac{\sum_{i=1}^{e} \sum_{j=1}^{e} |L_i - L_j|}{2n^2 \bar{L}} \quad (4)$$

where:

G is the Gini coefficient.

e is the number of endpoints.

L_i is the number of requests served by endpoint i.

$\bar{L}$ is the average number of requests across all endpoints.

V. Evaluation Setup

Three reward strategies have been considered in the evaluation of the *gym-loadbalancing* framework, as detailed in Table IV. The first strategy named *Latency* focuses on reducing both the processing and network latency, while the second strategy named *Inequality* targets a reduction of the Gini coefficient aiming to evenly distribute the requests across the available endpoints. Lastly, the third strategy denoted as *Multi-goal* focuses on both reducing the overall latency and promoting a fair distribution of requests with both corresponding weights set at 0.5.

Various RL algorithms have been assessed in the *gym-loadbalancing* environment. Most of these algorithms have been implemented based on the stable baselines 3 [28] library, a set of reliable implementations of RL algorithms written in Python. The evaluation consists mainly of four agents that support discrete action spaces:

- **Advantage Actor Critic (A2C)** [29]: A synchronous, deterministic algorithm that combines policy and value-based algorithms. Policy-based agents learn a policy mapping input states to output actions (i.e., actors), and value-based algorithms select actions based on the predicted value of the input state (i.e., critic).
- **Proximal Policy Optimization (PPO)** [30]: a policy gradient method for RL vastly used today for different scenarios (e.g., robot control and video games).
- **DeepSets PPO**: Similar to PPO but with a DeepSet as its neural network.
- **DeepSets Deep Q-Network (DQN)**: DQN combines the classical Q-Learning RL algorithm with deep neural networks. It also applies the DeepSet neural network.

Both DQN and PPO have been adapted to use the DeepSets neural network architecture by modifying their standard implementations in popular RL libraries. The policy network in PPO and the Q-network in DQN have been replaced with a DeepSet to assess its generalization capabilities.

The TeaStore application [31] has been used as a reference application to assess the performance of the RL algorithms. TeaStore is a widely used microservice benchmark framework consisting of seven workloads with distinct performance characteristics, allowing the evaluation of scheduling and load balancing techniques. It emulates a Web Store for automatically generated tea supplies and features several User interface (UI) elements for database generation and service resetting in addition to the store itself. We argue that TeaStore is a convenient application to test the proposed multi-zone load balancing approach since it consists of several microservices frequently sending requests across each other. Table V shows the TeaStore load balancing requirements applied in the evaluation.

The *gym-loadbalancing* framework has been implemented in Python to ease the interaction with both the OpenAI Gym and the stable baselines 3 libraries. In the evaluation, an episode consists of 100 steps where the RL agent attempts to maximize the accumulated reward based on the current request. If the agent selects one of the available endpoints to serve the request, both the endpoint processing latency (ϕ_e) and its CPU usage (Θ_{cpu}) increase. In contrast, if a request is terminated based on the mean service duration (as default one time unit), the processing latency and CPU usage decrease by decreasing the corresponding metrics. In addition, regarding the request, both the zone hosting the endpoint (Z_{in}) and the expected duration of the request (T_r) are randomized based on the number of available zones in the K8s cluster and the mean service duration, respectively. This ensures the RL agents receive different requests during training in consecutive episodes. During training, we used a cluster with four endpoints and four zones. The RL algorithms has been trained for 2000 episodes, a typical number used for RL training. The RL agents have been executed on 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:

- **Execution Time** of the given RL algorithm.
- **Accumulated reward** during each episode. It refers to the total sum of rewards obtained by an agent over time as it interacts with the *gym-loadbalancing* environment.
- **Percentage of rejected requests** represented as $[0, 100]$.
- **Average CPU usage** of the selected endpoint.

TABLE VI: The execution time during training.

Algorithm	Execution Time per episode (in s)	Execution Time for 2000 episodes
A2C	0.677 ± 0.007	22.56 minutes
PPO	0.476 ± 0.006	15.86 minutes
DeepSets PPO	0.897 ± 0.013	29.90 minutes
DeepSets DQN	0.576 ± 0.006	19.20 minutes
Topology-Greedy	0.114 ± 0.002	3.80 minutes
Zone-CPU-Greedy	0.132 ± 0.003	4.40 minutes
CPU-Greedy	0.115 ± 0.002	3.83 minutes

- **Average Processing latency** for each accepted request.
- **Average Network latency** for each accepted request.
- **Percentage of intra-zone and inter-zone traffic** represented as $[0, 100]$.
- **Gini Coefficient** highlighting the inequality of the load balancing strategy, represented as $[0, 1]$.

Three heuristic-based baselines have also been evaluated to compare against RL-based methods:

- **CPU-Greedy**: selects the endpoint with the lowest resource consumption (CPU usage).
- **Zone-Greedy**: chooses the endpoint from the zone with the highest CPU capacity based on its cluster nodes.
- **Topology-Greedy**: selects the endpoint providing the lowest network latency.

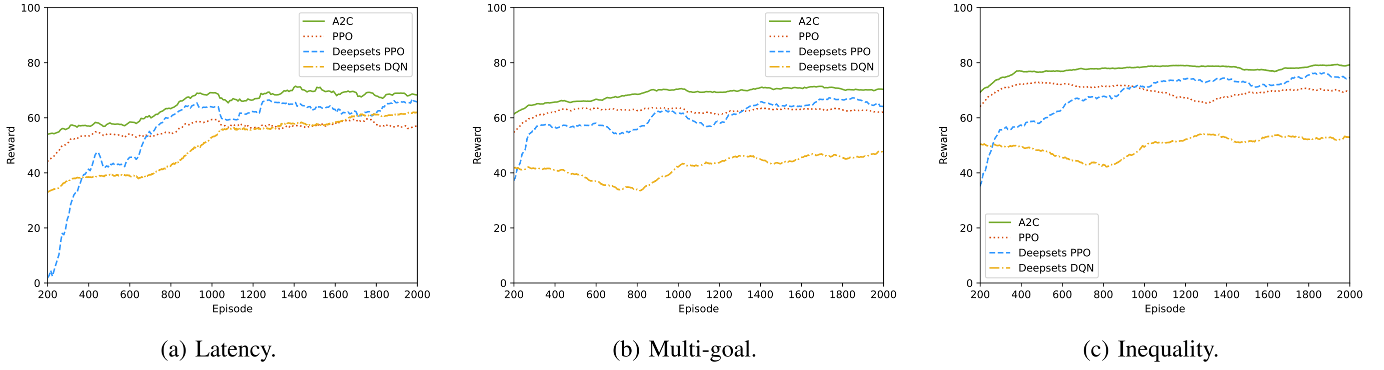
VI. Results

Time Complexity has been accessed based on the training execution time for the multiple RL agents for the latency-aware strategy (Table VI). The results highlight that training RL agents in near-real environments can speed up the applicability of RL methods in operational environments. Most algorithms require on average between 15 - 30 minutes for training during 2000 episodes. PPO is slightly faster than the other RL methods, especially compared to DeepSets PPO. All heuristics are significantly faster than most RL methods since no policy training is needed. On average, heuristic algorithms complete 2000 episodes within 3-4 minutes, whereas most RL algorithms typically require 15-23 minutes for the same task.

Training results for 2000 episodes are shown in Fig. 3 for all applied strategies. We applied a smoothing window of 200 episodes to mitigate spikes in the graphs. Despite fluctuations, all algorithms converge around the 1400th episode, even though some algorithms exhibit slight improvements in rewards beyond this point. All algorithms attain accumulated rewards surpassing 50 for the latency strategy, while demonstrating difficulties in achieving higher rewards for the Inequality strategy. This is particularly evident in the performance of the DeepSets DQN algorithm. It is noteworthy that all algorithms learn to minimize rejections, as evidenced by their pursuit of higher rewards. This trend will be further explored in our subsequent testing phase. Fig. 4 illustrates that all algorithms successfully learn to reduce the processing latency and the network latency during the training, achieving similar results to the Topology-Greedy approach. Also, Fig. 5 demonstrates that for the inequality strategy, latency is not the primary focus since average values are typically higher than the latency strategy, especially for the network latency.

Testing was done for all algorithms during 2000 episodes with the saved configuration after 2000 training episodes. Table VII summarizes the obtained results during the testing phase concerning the considered performance metrics for the different algorithms. Regarding the latency strategy, both DeepSets algorithms outperform A2C and PPO though the slightly worse training. However, these algorithms significantly reduce the expected total latency at a higher rejection rate of 2.5%. In fact, both algorithms achieve higher performance than the Topology-Greedy strategy since this primarily focuses on the network latency while neglecting the impact of the processing latency in the overall expected latency. To further assess the impact of rejection, the DeepSets algorithms (DeepSets PPO-R and DeepSets DQN-R) have been retrained without the option of rejecting requests by disabling it from the action space. The aim is to evaluate the performance of the algorithms when all requests need to be accepted (i.e., forwarded to a given endpoint). The final two rows of Table VII illustrate that the performance of DeepSets experiences a minor decrease without the rejection option, although the decline is not statistically significant. When rejection is permitted, these algorithms prefer to receive a penalty for 2.5% of the requests to slightly decrease the overall latency. For the multi-goal strategy, all algorithms demonstrate a tendency to reject fewer requests while attaining a slightly higher total latency, as expected. Additionally, most algorithms exhibit a lower *Gini coefficient*, which shows the effort toward a more equitable distribution of requests among the available endpoints. In the case of the inequality strategy, latency is considerably higher than the previous two strategies at a significantly lower *Gini coefficient*. Also, all algorithms exhibit a preference for inter-zone communication, diverging from the more balanced approach observed in the previous strategies. This result highlights that intra-zone traffic is preferred when latency is the primary goal. Furthermore, it is worth noting that all tested greedy approaches fail to provide a competitive alternative to RL methods, as these do not take into account the dynamics of the environment and the different aspects affecting the total latency of the requests. Among these approaches, the Topology-greedy method stands out as the only one capable of latency reduction, primarily by focusing on minimizing the network latency. Fig. 6 emphasizes this observation by presenting the CDFs for the various methods. As shown, it is evident that the total latency varies considerably across the learned policies of the RL algorithms, depending on the selected strategy and associated weights.

Generalization has been evaluated for both DeepSets algorithms by varying the number of available endpoints from 6 to 180. Fig. 7 demonstrates the enormous potential of the DeepSets neural network. Both algorithms can find adequate load balancing schemes for all strategies, even when initially trained in a small-scale scenario. As the number of endpoints increases, the complexity of the inequality strategy becomes notably more pronounced compared to the latency or multi-



(a) Latency. (b) Multi-goal. (c) Inequality.

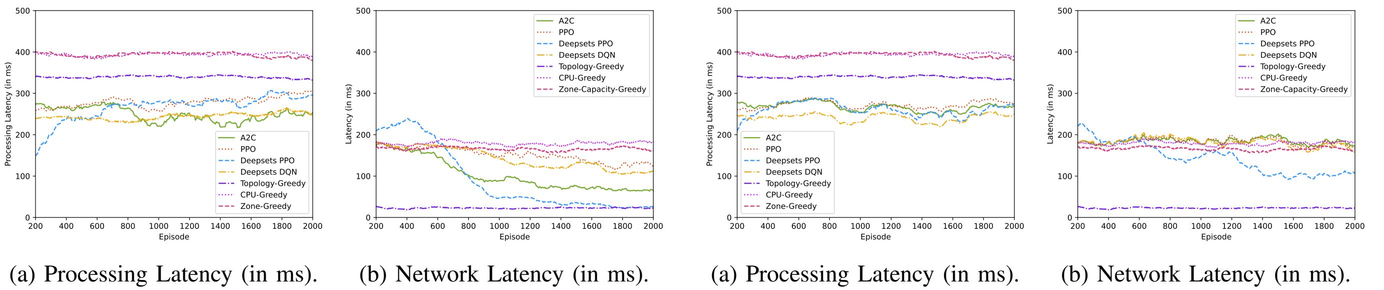
Fig. 3: The accumulated reward during training for the multiple agents.

TABLE VII: Results obtained during the testing phase for 2000 episodes.

Algorithm	Strategy	Rejected Requests (in %)	Processing Latency (in ms)	Network Latency (in ms)	Gini Coeff.	Inter-Zone Traffic (in %)	Intra-Zone Traffic (in %)
A2C	Latency	0.03 $\pm$ 0.007	**247.5** $\pm$ **2.9**	73.4 $\pm$ 1.53	0.28 $\pm$ 0.003	44.2 $\pm$ 0.6	55.8 $\pm$ 0.6
PPO	Latency	0.06 $\pm$ 0.01	297.9 $\pm$ 2.03	125.2 $\pm$ 2.03	0.48 $\pm$ 0.002	59.1 $\pm$ 0.5	40.9 $\pm$ 0.5
DeepSets PPO	Latency	2.5 $\pm$ 0.3	304.4 $\pm$ 1.8	**27.1** $\pm$ **0.9**	0.44 $\pm$ 0.005	26.9 $\pm$ 0.7	**70.5** $\pm$ **0.7**
DeepSets DQN	Latency	2.6 $\pm$ 0.3	**233.4** $\pm$ **4.23**	93.2 $\pm$ 1.89	0.37 $\pm$ 0.007	48.2 $\pm$ 0.6	49.2 $\pm$ 0.6
A2C	Multi-goal	0.01 $\pm$ 0.004	247.2 $\pm$ 2.9	74.5 $\pm$ 1.5	**0.25** $\pm$ **0.004**	44.8 $\pm$ 0.5	55.2 $\pm$ 0.5
PPO	Multi-goal	0.07 $\pm$ 0.01	278.7 $\pm$ 1.5	161.0 $\pm$ 2.1	0.28 $\pm$ 0.002	67.5 $\pm$ 0.3	32.4 $\pm$ 0.3
DeepSets PPO	Multi-goal	0.36 $\pm$ 0.07	282.08 $\pm$ 2.4	82.7 $\pm$ 1.5	0.38 $\pm$ 0.006	47.2 $\pm$ 0.6	52.4 $\pm$ 0.6
DeepSets DQN	Multi-goal	0.23 $\pm$ 0.04	269.7 $\pm$ 3.9	89.7 $\pm$ 2.2	0.49 $\pm$ 0.005	49.3 $\pm$ 0.6	50.5 $\pm$ 0.6
A2C	Inequality	0.006 $\pm$ 0.003	262.5 $\pm$ 1.7	184.1 $\pm$ 2.4	**0.13** $\pm$ **0.001**	72.5 $\pm$ 0.3	27.5 $\pm$ 0.3
PPO	Inequality	0.13 $\pm$ 0.01	273.8 $\pm$ 1.67	180.9 $\pm$ 2.4	0.26 $\pm$ 0.002	71.7 $\pm$ 0.3	28.1 $\pm$ 0.3
DeepSets PPO	Inequality	0.03 $\pm$ 0.01	270.8 $\pm$ 1.9	120.8 $\pm$ 1.5	0.33 $\pm$ 0.004	60.1 $\pm$ 0.7	39.8 $\pm$ 0.7
DeepSets DQN	Inequality	0.23 $\pm$ 0.05	287.6 $\pm$ 1.9	209.8 $\pm$ 3.5	0.50 $\pm$ 0.004	75.1 $\pm$ 0.5	24.7 $\pm$ 0.5
Topology-Greedy	-	0%	339.3 $\pm$ 2.06	**23.1** $\pm$ **1.4**	0.56 $\pm$ 0.004	16.4 $\pm$ 0.6	**83.6** $\pm$ **0.6**
Zone-Greedy	-	0%	392.9 $\pm$ 3.4	165.9 $\pm$ 2.73	0.83 $\pm$ 4.8	66.1 $\pm$ 0.4	33.9 $\pm$ 0.4
CPU-Greedy	-	0%	393.4 $\pm$ 3.3	179.4 $\pm$ 3.02	0.83 $\pm$ 9.79	71.8 $\pm$ 0.4	28.17 $\pm$ 0.4
DeepSets PPO-R	Latency	0%	295.4 $\pm$ 2.4	29.3 $\pm$ 1.5	0.48 $\pm$ 0.005	25.3 $\pm$ 0.7	74.7 $\pm$ 0.7
DeepSets DQN-R	Latency	0%	240.4 $\pm$ 4.6	95.5 $\pm$ 2.1	0.44 $\pm$ 0.007	52.7 $\pm$ 0.6	47.3 $\pm$ 0.6

TABLE VIII: The average network latency for each request increases, if network costs are not updated.

	Network Latency (in ms)					Total Latency (in ms)			
Algorithm	**+0%**	**+10%**	**+30%**	**+50%**	**+70%**	**+10%**	**+30%**	**+50%**	**+70%**
DeepSets PPO	**27.1**	40.7 $\pm$ 0.9	47.9 $\pm$ 1.6	55.3 $\pm$ 1.9	62.5 $\pm$ 2.1	348.3 $\pm$ 2.2	355.6 $\pm$ 2.4	362.8 $\pm$ 2.5	370.1 $\pm$ 2.7
Topology-Greedy	**23.1**	33.2 $\pm$ 1.3	35.9 $\pm$ 2.0	41.3 $\pm$ 2.3	46.7 $\pm$ 2.6	372.1 $\pm$ 2.2	374.8 $\pm$ 3.2	380.1 $\pm$ 3.5	385.5 $\pm$ 3.7



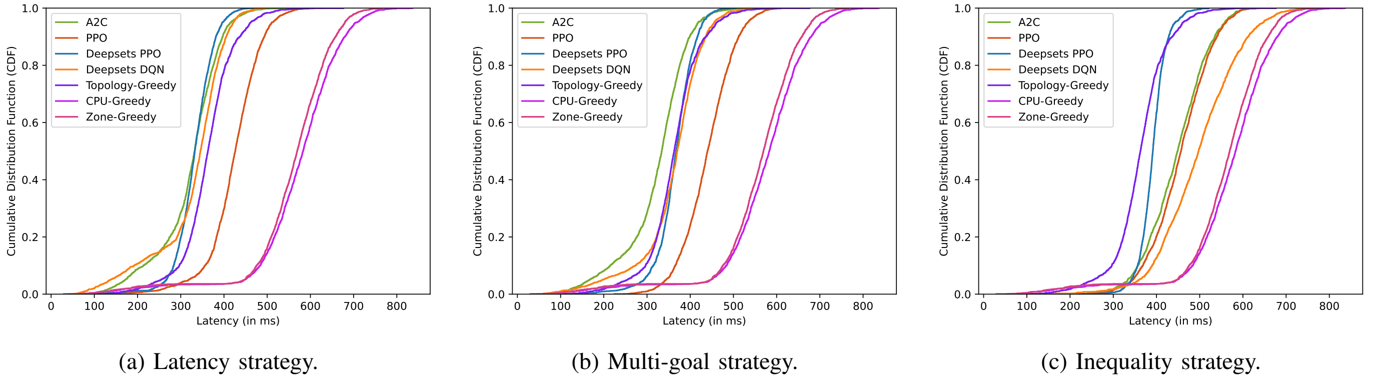
(a) Processing Latency (in ms). (b) Network Latency (in ms).

Fig. 4: The expected latency for the multiple agents during training for the latency strategy.

(a) Processing Latency (in ms). (b) Network Latency (in ms).

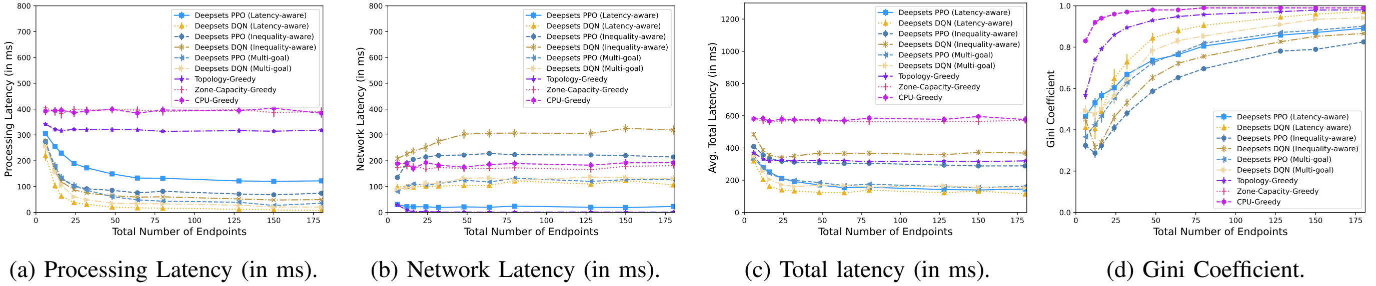
Fig. 5: The expected latency for the multiple agents during training for the inequality strategy.

goal strategies. The *Gini coefficient* increases throughout the experiment, showing how difficult it is to find an even distribution of requests among the available endpoints. Also, both algorithms exhibit the ability to maintain both the processing and network latency at a nearly constant level as the number of endpoints increases, showing the potential of these algorithms in being applied in operational environments without requiring retraining. It is also noteworthy that both agents exhibit differences in their learned policy since DQN favors the reduction of the processing latency, while PPO prefers to minimize the network latency, though achieving similar results for the total latency. In conclusion, both algorithms can provide efficient load balancing schemes in a multi-zone K8s cluster 30 times higher than its initial trained setup.



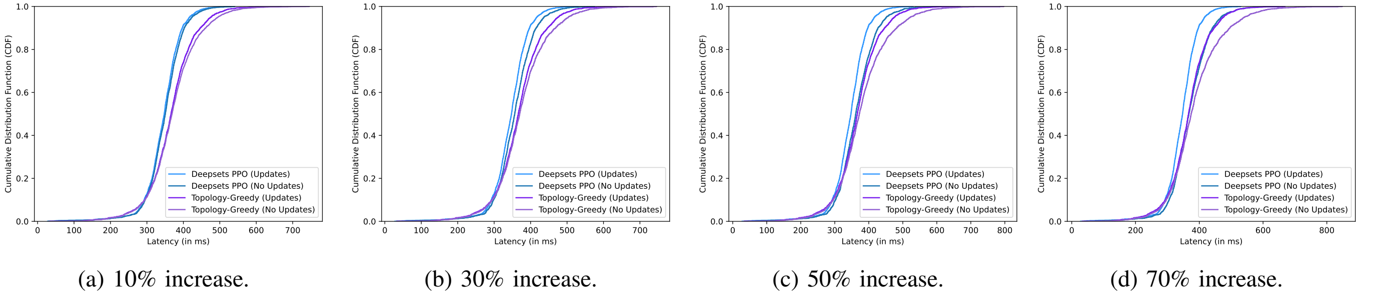
(a) Latency strategy. (b) Multi-goal strategy. (c) Inequality strategy.

Fig. 6: The Cumulative Distribution Function (CDF) for the total latency during testing for the multiple evaluated algorithms.



(a) Processing Latency (in ms). (b) Network Latency (in ms). (c) Total latency (in ms). (d) Gini Coefficient.

Fig. 7: The results for the trained DeepSets agents for all strategies while varying the number of available endpoints.



(a) 10% increase. (b) 30% increase. (c) 50% increase. (d) 70% increase.

Fig. 8: The CDF for the total latency when network costs are increased and updates are not communicated to the agent.

What if network costs are not updated? As a final step in our evaluation, this study examines the implications of unavailable or infrequently updated network costs in a typical multi-zone K8s cluster. In this analysis, both DeepSets PPO and the Topology-Greedy strategy have been evaluated over 2000 episodes, considering predetermined increases in the range of $[+10\%, +30\%, +50\%, +70\%]$. Table VIII indicates a significant increase in network latency when network costs are not updated. For instance, with a 10% increase, DeepSets PPO achieves an average network latency of 40.7 ms instead of 27.1 ms, representing a 0.5x increase, while the Topology-Greedy approach obtains an average network latency of 33.2 ms instead of 23.1 ms, representing a 0.43x increase. Both algorithms experience a twofold increase in network latency when network costs are increased by 70%. Fig 8 illustrates the CDF for the total latency across all scenarios, revealing significant differences for both algorithms when network costs increase beyond 30%. These findings highlight the critical importance of accurate network costs when implementing such approaches in typical K8s clusters. This can be achieved currently within a K8s cluster by utilizing the previously presented NetworkTopology CRD.

In summary, this paper investigates efficient multi-zone load balancing strategies tailored for the widely-used K8s platform, focused on recent trends such as RL. Through the exploration of various competing strategies, it becomes evident that RL algorithms have the capacity to discern optimal actions that maximize accumulated rewards based on predefined objectives. Validation through an RL environment reaffirms the efficacy of this approach, with most algorithms significantly outperforming traditional greedy methods. The adoption of the proposed RL-based approach could enhance the current load balancing mechanism within K8s by finding an appropriate balance between latency reduction and equitable request distribution, as evidenced by the obtained performance during testing. Moreover, the DeepSets neural network emerges as a pivotal component, showcasing immense potential. These RL algorithms can seamlessly adapt to diverse

multi-zone environments with varying endpoints, thereby mitigating the need for extensive retraining. Without the utilization of DeepSets, the retraining of RL algorithms for specific endpoint configurations would entail considerable execution time and computing resources, limiting the efficiency gains facilitated by its integration.

VII. Conclusions

This paper studies the challenge of efficient load balancing of microservices within a multi-zone cluster infrastructure. An RL-based approach inspired on the OpenAI Gym library has been proposed to enable efficient load balancing in the well-known K8s platform. The evaluation considers three strategies that show the feasibility of RL for the microservice load balancing problem addressed in the paper. The results show that generalization is attainable by incorporating the DeepSets neural network in typical RL algorithms, achieving high performance even in scenarios up to 30 times larger than the trained environment. Multi-agent RL scenarios will be studied as future work to find optimal combinations of opposing scheduling strategies. In addition, we plan to integrate our RL approach into the K8s ecosystem in the form of plugins to compare our approach with existing load balancing mechanisms. Moreover, our work contributes to the field by providing a framework released in open-source, allowing researchers to evaluate load balancing concepts and potentially guide the development of more efficient load balancing algorithms.

Acknowledgment

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

References

[1] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[2] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.

[3] B. Burns, J. Beda, and K. Hightower, *Kubernetes: up and running: dive into the future of infrastructure*. O'Reilly Media, 2019.

[4] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 198–206.

[5] S. Afzal and G. Kavitha, "Load balancing in cloud computing–a hierarchical taxonomical classification," *Journal of Cloud Computing*, vol. 8, no. 1, p. 22, 2019.

[6] 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.

[7] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures using microservices: an experience report," in *Advances in Service-Oriented and Cloud Computing: Workshops of ESOCC 2015, Taormina, Italy, September 15-17, 2015, Revised Selected Papers 4*. Springer, 2016, pp. 201–215.

[8] J. Santos, J. van der Hooft, M. T. Vega, T. Wauters, B. Volckaert, and F. De Turck, "Srfog: A flexible architecture for virtual reality content delivery through fog computing and segment routing," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2021, pp. 1038–1043.

[9] 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.

[10] M. Autili, A. Perucci, and L. De Lauretis, "A hybrid approach to microservices load balancing," *Microservices: Science and Engineering*, pp. 249–269, 2020.

[11] R. Yu, V. T. Kilari, G. Xue, and D. Yang, "Load balancing for interdependent iot microservices," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 298–306.

[12] Y. Liang and Y. Lan, "Tclbm: A task chain-based load balancing algorithm for microservices," *Tsinghua Science and Technology*, vol. 26, no. 3, pp. 251–258, 2020.

[13] F. Wan, X. Wu, and Q. Zhang, "Chain-oriented load balancing in microservice system," in *2020 World Conference on Computing and Communication Technologies (WCCCT)*. IEEE, 2020, pp. 10–14.

[14] H. Zhu, H. Wang, and I. Bayley, "Formal analysis of load balancing in microservices with scenario calculus," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 908–911.

[15] M. M. Khalel, M. A. Pugazhendhi, and G. R. Raj, "Enhanced load balancing in kubernetes cluster by minikube," in *2022 International Conference on Smart Technologies and Systems for Next Generation Computing (ICSTSN)*. IEEE, 2022, pp. 1–5.

[16] K. Takahashi, K. Aida, T. Tanjo, and J. Sun, "A portable load balancer for kubernetes cluster," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2018, pp. 222–231.

[17] N. Nguyen and T. Kim, "Toward highly scalable load balancing in kubernetes clusters," *IEEE Communications Magazine*, vol. 58, no. 7, pp. 78–83, 2020.

[18] G. Sayfan, *Mastering kubernetes*. Packt Publishing Ltd, 2017.

[19] L. Calcote and Z. Butcher, *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*. O'Reilly Media, 2019.

[20] A. Khatri and V. Khatri, *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*. Packt Publishing Ltd, 2020.

[21] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-aware scheduling in container-based clouds," *IEEE Transactions on Network and Service Management*, 2023.

[22] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[23] O. Mart, C. Negru, F. Pop, and A. Castiglione, "Observability in kubernetes cluster: Automatic anomalies detection using prometheus," in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2020, pp. 565–570.

[24] N. D. 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)*, 2023, pp. 125–133.

[25] J. Santos, M. Zaccarini, F. Poltronieri, M. Tortonesi, C. Stefanelli, N. Di Cicco, and F. De Turck, "Efficient microservice deployment in kubernetes multi-clusters through reinforcement learning," in *NOMS 2024-2024 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2024, pp. 1–9.

[26] Amazon AWS, "Amazon ec2 on-demand pricing." accessed on 28 September 2023. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/.

[27] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "Demand-driven deep reinforcement learning for scalable fog and service placement," *IEEE Transactions on Services Computing*, vol. 15, no. 5, pp. 2671–2684, 2021.

[28] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3," 2019.

[29] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[30] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "Implementation matters in deep rl: A case study on ppo and trpo," in *International conference on learning representations*, 2019.

[31] J. Von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "Teastore: A micro-service reference application for benchmarking, modeling and resource management research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018, pp. 223–236.