Scheduling Latency-Sensitive Tasks in the Cloud Continuum with Hierarchical Reinforcement Learning

Scheduling Latency-Sensitive Tasks in the Cloud Continuum with Hierarchical Reinforcement Learning

Doriana Monaco * Alessio Sacco * Claudio Casetti * Guido Marchetto *
* *Department of Control and Computer Engineering*, Politecnico di Torino, Italy

***Abstract*—Service orchestrators such as Kubernetes are widely employed to automate the handling and scheduling of workloads, which involves determining the most suitable physical node on which to start a new task. The expanding application of Machine Learning (ML) algorithms, and in particular Reinforcement Learning (RL), opens up new development opportunities to make runtime decisions that can account for multiple metrics and varying network conditions. However, current RL-based solutions are unable to fit the growing complexity of distributed applications and infrastructure, characterized by a more heterogeneous resource continuum and the increasing need to minimize energy consumption while satisfying tasks' requirements. To fill this gap, we propose RL-ICE as an innovative scheduler that can work in such a cloud continuum by leveraging a multi-cluster and hierarchical RL to satisfy both user Quality of Experience (QoE) metrics and tenant's costs. We test RL-ICE in a simulated large-scale environment and in a real-world Kubernetes setup. In both scenarios, our solution effectively balances user-perceived latency, energy consumption, and deployment costs. Additionally, RL-ICE can dynamically respond to network failures by migrating microservices to maintain efficient management of resources.**

***Index Terms*—Kubernetes, Scheduling, Reinforcement Learning**

I. Introduction

In recent years, containerization has significantly transformed the deployment and management of applications [1]–[3]. By allowing applications to be packaged in a lightweight and interoperable format, containers have enabled a shift from monolithic architectures to microservice-based systems, greatly enhancing deployment flexibility and operational efficiency. This transformation has been central to the cloud-native revolution, where applications are distributed across entire data centers and managed by dedicated orchestrators. Among these, Kubernetes has become the de facto open-source platform for container orchestration, efficiently managing the entire deployment lifecycle, from scheduling and scaling to self-healing and descheduling of pods, the smallest deployable units [4].

However, managing these microservice-based applications introduces complexity, particularly with the rise of new stringent scenarios– IoT, autonomous vehicles– that demand low latency and flexible deployments. In this context, Edge and Fog paradigms gain prominence addressing user proximity, low latency and privacy issues [5]–[7], through the distribution of multiple smaller data centers at the network edge. This scenario paves the path for the development of the edge-cloud continuum, which aim to manage a comprehensive set of geographically distributed resources as a unique virtual space.

A key component of microservices orchestration is the scheduler, which undertakes the complex task of organizing and executing the distribution of pods across nodes. While Kubernetes excels in managing resources in traditional cloud environments, primarily optimizing CPU and memory resources [8], default scheduling mechanisms are less suited for edge-cloud environments, where network latency and proximity to users have a critical impact on application performance. In response, new orchestration solutions are being developed to optimize the distribution of containerized applications across geo-distributed and edge-cloud continuum infrastructures [9]–[11], focusing on multi-cluster environments. These solutions aim to enhance service agility by incorporating real-time network metrics and proximity considerations into scheduling decisions. Recent efforts leverage Machine Learning (ML) techniques, especially Reinforcement Learning (RL), to deploy scheduling solutions that can dynamically adapt to changing conditions [12]–[17]. Nonetheless, some studies focus solely on efficient resource utilization [12], [14] or system latency minimization [13], and only a limited fraction of research works has addressed the reduction of network latency [15]–[17]. Despite the soundness of these approaches, current solutions do not consider user-specified intents and real perceived latency in the edge-cloud continuum.

In this paper we propose **R**einforcement **L**earning for **I**ntent-driven **C**loud-**E**dge scheduling (RL-ICE), a RL-based scheduler that well fits this scenario with a hierarchical approach where one model (outer model) identifies the optimal cluster and another (inner model) selects the best node within it. Our system is designed to meet user intents by accommodating latency requirements while also considering other administrative objectives such as cost optimization and energy efficiency. To effectively balance these multiple objectives, we take into account both node metrics and user-perceived latency. Additionally, both type of models can handle a varying number of input–cluster or nodes– by adopting a DeepSet approach [18], thus making our solution well adapting to large-scale scenarios or failures, ensuring efficient and sustainable resource management.

We evaluate RL-ICE against various benchmarks, both learning-based and non-learning-based schedulers. We first test its effectiveness and scalability in a simulated large-scale

environment, where it outperforms state-of-the-art in balancing multiple objectives and also shows its ability to dynamically adapt to user movement and network failures by migrating microservices to meet the specified requirements. Finally, we deploy it in Kubernetes prototype and compare it to the default Kubernetes scheduler and other available policies. In this setup, RL-ICE outperforms the benchmarks in minimizing the latency while reducing the number of used machines.

The rest of the paper is structured as follows. We first review state-of-the-art scheduler proposals in Section II. Subsequently, we define the Cloud-Edge Continuum and motivate our solution in Section III and describe in detail the RL model in Section IV. Later on, we propose a possible implementation in the Kubernetes environment in Section V and evaluate the overall system in Section VI. Lastly, we draw the conclusions of our work in Section VII.

II. Related work

Kubernetes (K8s), the most widely used system to schedule containerized applications, entails a central controller (running on a master node) responsible for deploying tasks as they arrive. This activity occurs with a system-wide perspective to enhance optimization, but it primarily focuses on optimizing computational resources utilization [4]. Several recent proposals attempted to enhance the default scheduling decisions by means of user-aware or network-aware metrics encompassing the communication demands of modern latency-sensitive applications, from which the Cloud-Edge Continuum would greatly benefit.

Since the scheduling problem is NP-Hard [4], heuristics constitute a simple and satisfactory approach to adopt. Some works optimize the application traffic among nodes [19]–[23], while others minimize inter-node latency [24], [25]. Some research studies focus on intra-cluster scheduling [14], [26], while others analyze a multi-cluster scenario [17], [27], [28]. For example, authors in [26] propose a network-aware greedy heuristic to optimize the placement of pods in a geo-distributed environment. This heuristic considers the deployment time, available computing resources, and network delays between worker nodes. PIVOT [27] adopts an heuristic algorithm to schedule tasks on geo-distributed VMs to minimize cloud costs and improving data transfer efficiency. They take into account VM and network traffic costs but also data locality and network bandwidth to place tasks close to their data sources. Mck8s [28] proposes a multi-cluster orchestrator that provides either resource-based (worst-fit and best-fit) or network traffic based (traffic-aware) placement policies based on user-defined preferences.

In recent years, Machine Learning (ML) has emerged as an alternative to traditional heuristic approaches, where sophisticated models are used to make efficient scheduling decisions by learning from the data they are provided [12]–[17], [29]. In particular, the success of Reinforcement Learning (RL) is due to its closed-loop training process, which involves an agent interacting with and learning from a multi-cluster environment. Some preliminary results are presented in RLKube [14], where many RL models with different objectives (e.g., maximize resource utilization, enhance Pod throughput, improve energy efficiency) are evaluated against the K8s default policy Least-Allocated (K8s-LA) and the Most-Allocated (K8s-MA) plugin. The experiments show that most of the RL policies outperform the non-learning ones. DRL-FORCH [16] leverages DRL to optimize a bi-objective reward when selecting the appropriate fog node. The goal is to minimize the blocking probability and satisfy the latency QoS requirements of each service. Through the adoption of a DeepSet neural network, they achieve good performance across settings with varying number of fog nodes. [17] focuses on the distribution of replicas in a multi-cluster environment, proposing a cost-aware model and a latency-aware model. Based on clusters characteristics and deployment requirements, the RL models decide to either deploy all the replicas in a cluster or spread them over many. Later on, a First Fit Decreasing (FFD) algorithm determines the specific clusters. Both models experimentally outperform greedy algorithms and generalize over a varying number of clusters thanks to the adoption of DeepSets.

Despite this multitude of research works that aim at minimizing the latency in cloud environemnts, only a few of them focus on user-perceived latency [11], [30], [31]. Unlike existing solutions, our approach is tailored for multi-cluster cloud-edge environments, employing a hierarchical strategy for intelligent scheduling powered by RL. In this design, one model is responsible for selecting the optimal cluster, while another operates within the cluster to choose the best node. This dual-layered approach enables us to efficiently balance multiple objectives: lowering tenant costs, fulfilling user-defined latency intents, and maximizing energy efficiency. Furthermore, to facilitate the training procedure and the portability of the model, we specifically design a model that can adapt to settings that are diverse from training (e.g., large-scale scenarios), thus leading to robust, efficient, and sustainable resource management.

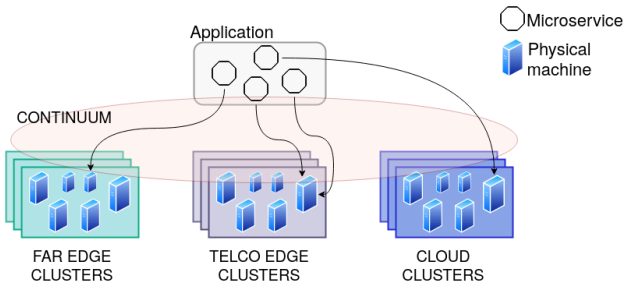


Fig. 1: Performing task scheduling in the Cloud Continuum.

III. System overview

In this section, we outline our perspective on the cloud-edge continuum, discussing its key characteristics, benefits, and challenges. This provides the rationale behind the design decisions of our scheduler, which we introduce afterward.

The idea of providing unified computing resources within a single continuum has been around for a few years, with multiple solutions proposed both from the scientific community and

enterprise-driven projects [9], [32]. The key idea is to present a cohesive perspective of numerous distributed clusters. In this paper, we specifically consider physically distributed machines both in the cloud and closer to users, i.e., at the edge. In this context, the clusters that want to join the continuum coexist as peers and share their resources. We assume that the tenants that provide visibility over their clusters announce their resources through a resource management protocol, such as the REAR [33] protocol that allows announcing and reserving computing resources.

As depicted in Fig. 1, the computing continuum is a set of distributed resources that are envisioned as a unique virtual space, on top of which applications operate. In this way, services can leverage all the resources within the virtual space, regardless of their physical location. Moreover, the communication between two services that belong to the same virtual space occurs as if they were part of the same cluster. Although resources are shared within the continuum, each tenant is always fully in charge of its own infrastructure and can decide to join the continuum or leave in any moment. The orchestrator managing the continuum serves as an abstraction layer, relieving users from the need to understand or control the physical infrastructure where their applications are executed. Instead, users can define performance requirements or constraints as intents, and the scheduler optimizes resource allocation to ensure these intents are met efficiently.

The creation of a cloud-to-edge continuum, however, requires several complex techniques to be coordinated at several levels.

- **Dynamic environment**: peers of the continuum can join and leave at any time, creating a highly dynamic environment to orchestrate, augmenting or reducing the number of resources available.
- **Large-scale and heterogeneous continuum**: Managing large-scale, geographically distributed environments is inherently complex. The diverse nature of computing resources, as well as the fluctuating demand and network performance, requires real-time decision-making to optimize resource allocation.
- **User Intent Alignment**: Accurately understanding and aligning resource allocation with user intent is a significant challenge. Scheduling decisions must strike a balance by fulfilling user requirements while also maintaining other critical performance metrics, ensuring that optimizing for one goal does not negatively impact others.
- **Energy efficiency**: Physical machines, both in cloud data centers and at the edge, consume significant energy. Balancing energy consumption and high performance is thus considered a major challenge from infrastructure owners.
- **Cost Optimization**: Achieving cost efficiency for tenants across a cloud-to-edge continuum is challenging due to the dynamic nature of resource pricing, varying infrastructure costs across regions, and fluctuating demand. Effectively managing these variables without compromising performance requires sophisticated cost-aware strategies.

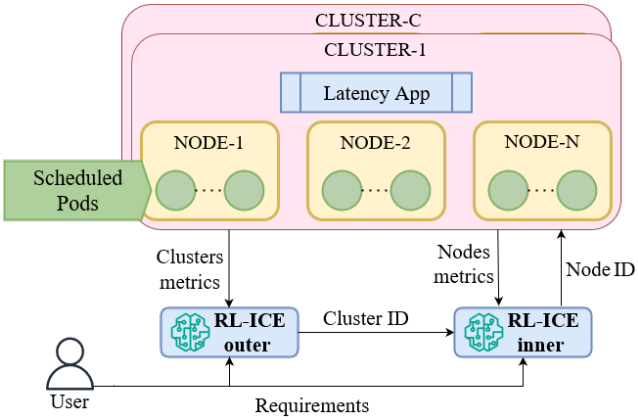


Fig. 2: Overall system architecture. To meet cloud continuum characteristics, RL-ICE comprises two layers of RL models: an outer and an inner model.

To address these challenges, we introduce Reinforcement Learning for Intent-Driven Cloud-Edge Scheduling (RL-ICE), a multi-cluster scheduler designed to incorporate user-defined requirements into its decision-making process. This approach leverages hierarchical reinforcement learning (RL) to effectively manage geographically distributed and heterogeneous resources, optimizing the trade-offs among various objectives while facilitating scheduling within the cloud-edge continuum. Users can assign constraints through high-level policies, ignoring the details about the physical infrastructure. Our solution involves two levels of decision-making: an outer scheduler selects a cluster from the virtual space that has sufficient resources, balancing user-perceived latency with tenant costs. For intra-cluster allocation, an inner scheduler determines the optimal execution location, minimizing machine costs and energy consumption. This approach supports the scenarios where certain tenants prefer to keep detailed information about their resources private: in such cases, they choose the node internally without relying on our RL model and only the outer model is executed. Moreover, we deploy a scalable model that, through the use of DeepSets (Section IV-C), can adapt to dynamic changes in the environment.

We present the system architecture in Fig. 2. The user can specify some latency requirements in the form of intent. In particular, it can express a *hard constraint*, which identifies a latency threshold to strictly respect, and a *soft constraint*, a threshold to possibly satisfy. These intents will be leveraged by RL-ICE to make informed decisions regarding the placement of each pod replica [34]. A latency measurement tool running on each cluster collects real-time, user-perceived latency on-demand. This data, combined with the user's latency requirements and tenant ID, forms the first input for the decision-making model. The outer model selects the appropriate cluster for deployment and informs the inner model. Using the cluster ID, along with the pod's resource requirements and the available resources on each node, the inner model determines the specific node where the pod replica will be scheduled.

IV. HIERARCHICAL RL SCHEDULER

We now explore the details of our RL-based scheduler, highlighting its components and how they interact with the environment. Additionally, we will explain the rationale behind our design choices, particularly the decision to utilize reinforcement learning.

Given the computational complexity associated with solving scheduling problems in large networks, RL approaches are being successfully adopted [12]–[17]. This methodology excels due to its trial-and-error framework, allowing the agent to learn effectively without requiring prior knowledge of the system dynamics. In fact, the agent undergoes a training phase where it interacts with the environment and collects the current state of resources. Based on that, it chooses an action to perform and this is rewarded or penalized based on its effectiveness. After undergoing a comprehensive training process, the RL agent can be deployed in real networks, enabling it to make faster and more efficient decisions compared to traditional methods.

To comply with the scenario described in Section III, we chose to deploy two different models. An outer model is responsible for the selection of one of the available clusters. If the peers provide visibility over their resources, as envisioned in modern continuum [33], the system can leverage an inner model to optimize the node selection to allocate the pod replica. These models operate in a single step, with one model informing the other and the problem is treated as a Markov Decision Process (MDP). Each MDP is defined by a quadruple $M = \langle S, A, P, R \rangle$, where S is the state space, A is the action space, P represents the probability to transition from a state s to a state s', R is the reward function. Each model is trained using the Deep Q-Learning algorithm and two neural networks: a *main* network parametrized by θ and a *target* network parametrized by θ'. By updating these two networks at different frequencies, we enhance training stability. During training, the model objective is to minimize the loss function that defined is the following way:

$$Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2, \tag{1}$$

where $s, s' \in S$, $a, a' \in A$, $r \in R$, Q is the action-value and represents the likelihood of taking each action, and γ is the discount factor and determines the importance of future rewards.

This hierarchical system benefits from several key features:

- **Reduction in Complexity**: The use of smaller, more focused models helps simplify the problem and accelerate the learning phase.
- **Objective specialization**: The hierarchical system employs two distinct models that concentrate on different objectives. This specialization allows for more effective problem-solving, as each model can be fine-tuned to address specific goals.
- **Subtask Reuse**: Similar subtasks can be applied in different contexts, improving efficiency and reducing the need for repeated learning. A single generalizable model for node selection is trained and deployed across different clusters.

A. MDP formulation of the outer model for cluster selection

We utilize RL-ICE to choose the appropriate location for each pod replica that must be deployed. In particular, the outer model is in charge of choosing the appropriate cluster of deployment, following an RL-based formalization with the following variables:

State space. We model the peer clusters as a dynamic set, such that if a tenant wants to leave or join the continuum, it is removed or added to the set. Each cluster belongs to a tenant and is a data center constituted by co-located physical machines that are shared with peer clusters in other regions. For this reason, for each user that performs a deployment request, we characterize the clusters with a tenant id and a latency value measured through a ping between one of the nodes and the user. Along with these clusters features, we consider the latency requirements, i.e., hard and soft constraints, of the pod replica to be scheduled. Therefore, the state space dimension depends on the number of peer clusters, referred to as C, their features and the two constraints.

Action space. The action space is a vector of length C containing the probability distribution of choosing each cluster. Even though the size of the action space linearly grows with the number of clusters, we advocate that the number of clusters in large scale environments remains limited as they are aggregation points. Additionally, we leverage action masking to reduce the possible actions in a step, e.g., by setting the action-value of clusters without enough resources to $-\infty$. Action masking is very effective in improving convergence time and performance, as demonstrated in previous works [35]–[38].

Reward. The cluster model objective is to satisfy the user-specified latency while reducing the resource provider costs. In fact, from a service provider's viewpoint, the external resources leased by other peers are costly compared to the owned ones, hence their usage should be reduced. We design a reward function resembling this objective.

We indicate the perceived latency between the user requesting the deployment and the chosen cluster with *cls_lat* and construct the reward function with a latency term:

$$R_l = \begin{cases} 1, \text{if cls\_lat} < \text{soft\_constraint} \\ 0.5, \text{if soft\_constraint} < \text{cls\_lat} < \text{hard\_constraint} \\ 0, \text{otherwise} \end{cases} \tag{2}$$

and a penalty term:

$$R_t = \begin{cases} 1, \text{if the cluster owner is another tenant} \\ 0, \text{otherwise} \end{cases} \tag{3}$$

such that the final reward equation is:

$$R_c = R_l - \alpha R_t \tag{4}$$

where $\alpha \in \{0, 1\}$ is a weight factor to balance the objectives.

B. MDP Formulation of the inner model for node selection

Once the cluster is defined, for those tenants that provide visibility on their resources, we leverage an inner model to select the physical machine.

State space. We model the nodes inside each cluster as a dynamic set, such that at each step we consider possible failures or additional machines instantiated. Each node is characterized by used and total resources in terms of CPU, RAM and Bandwidth but also by its price, as reported in Table II. The state space is therefore dependent on the number of nodes in the cluster, denoted as N, their features and the resources requested by each pod replica.

Action space. The action space is a vector of length N containing the probability distribution of choosing each node. To minimize the explosion of the action space, we leverage action masking to inform the model about which nodes have enough resources, while the model focuses on the best one to optimize the reward.

Reward. The node model is responsible for an allocation scheme that efficiently utilizes the available machines, such that as many pod replicas as possible are accommodated, while minimizing the energy consumption and the deployment cost. The reward signal after choosing node i is:

$$R_i = \frac{\text{\#Pods\_allocated}}{\text{\#Nodes\_on}} - \beta cost \tag{5}$$

where cost is defined as:

$$cost = \begin{cases} \frac{price_i}{max\_price}, \text{if } i \text{ is an idle node} \\ 0, \text{if } i \text{ already hosts some pods} \end{cases} \tag{6}$$

- The first term increases the energy efficiency and the throughput, encouraging allocation scheme that fits more pods in less nodes.
- The second term minimizes the operator expenses, penalizing the use of machines with a higher price.
- $\beta \in \{0, 1\}$ is a weight factor to balance the objectives.

We design this model to be generalizable across different clusters. To achieve this, we leverage the DeepSet architecture, which is used also to implement dynamic input sets.

C. Generalization with Deepsets

Neural networks are typically designed to process fixed-dimensional input and output data, but many real-world problems involve inputs that can vary in size, e.g., due to add or removal of new computing nodes. To address this, recent advancements in machine learning have focused on extending algorithms to handle inputs and outputs that are sets, which are inherently permutation-invariant. These inputs lack a specific order, but their overall structure is essential.

In this context, we employ the DeepSets [18] architecture, which incorporates equivariant transformations to process such data. This architecture ensures equivariance to permutations, meaning if the input elements are rearranged, the output reflects this same permutation. In contrast, invariance implies the output remains the same regardless of input order. The use of equivariant functions is crucial in our model because we require a direct mapping between the inputs and the corresponding outputs, such as selecting nodes based on Q-values. Since the composition of multiple equivariant functions also results in an equivariant function, DeepSets can be constructed by layering such functions, ensuring robust handling of unordered input data.

Given an input set $x \in \mathbb{R}^{n \times m}$, consisting of n elements each with m features, a permutation-equivariant neural network layer $f(x) : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times k}$ is defined as follows:

$$f(x) = \sigma(\Lambda \mathbf{x} - 1^T \Gamma maxpool(\mathbf{x})) \tag{7}$$

where $\Lambda, \Gamma \in \mathbb{R}^{m \times k}$ are trainable parameters and $\sigma(\cdot)$ is a non-linear activation function, i.e. ReLU, and *maxpool* extracts the maximum value of each feature column across all elements in the set. This operation is permutation-equivariavant with respect to the rows of $\mathbf{x}$, as *maxpool* is a permutation-invariant operation and, later on, additional transformations are performed to each row. Additionally, these operations do not rely on a fixed number of elements, allowing Deep Sets Neural Networks to perform inference on input sets of varying size.

In the context of DQN algorithms, permutation-equivariant Deep Sets can be utilized to define a policy $\pi_\theta(a|s) : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^n$ for a discrete action space across the elements of a set. Here, the state $s \in \mathbb{R}^{n \times m}$ represents a collection of n elements, each characterized by m features. The action $a \in \mathbb{R}^n$ corresponds to a categorical distribution over these n elements in the input set.

In our hierarchical model, we employ the DeepSets architecture for both the inner and outer models. By applying this to the outer model (cluster selection), the decision-making process becomes independent of the input set. This means it can still work even if certain clusters are temporarily unavailable and should be excluded from consideration for incoming requests. Similarly for the inner model (node selection), where our schema can manage circumstances in which nodes vary in cardinality. In addition, this approach enables us to train the model just once and utilize it for multiple clusters. As a result, we achieve a more precise model and reduce the overhead, since the scheduler only needs two models (i.e., one outer and one inner model) even when managing numerous clusters, as we can re-use the inner model for various clusters. As a result, the two neural networks are composed of three equivariant layers followed by ReLU activation except for the last layer, which is squeezed to obtain an array of scalar Q-values.

V. K8S PROTOTYPE

Kubernetes is widely recognized for its flexibility and extensibility, making it an ideal platform for handling complex orchestration tasks [2], [4]. With this in mind, we prototyped our solution for seamless integration into Kubernetes. We leverage Liqo [39] to enable the communication among clusters and develop a Golang plugin that integrates smoothly into the Kubernetes architecture, ensuring compatibility with future updates in the ecosystem. This approach enables us to benefit from Kubernetes' reliable and well-established default scheduler while extending its functionality to meet the specific needs of our use case. To thoroughly study and evaluate its functionality, all default plugins have been disabled. This

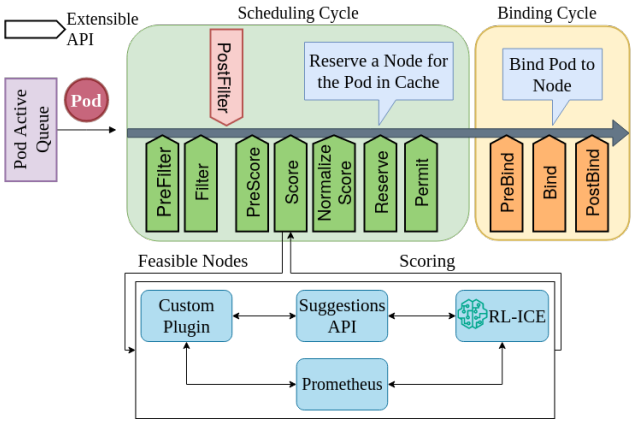


Fig. 3: Custom k8s scheduler plugin

leaves only the custom plugin active, ensuring that its effects on the scheduling process can be observed in isolation without interference from other default scheduling behaviors. However, as we deployed it as an additional plugin, RL-ICE can also work with existing Kubernetes plugins, e.g., HPA for horizontal and vertical autoscaling [34].

Indeed, our plugin specifically intervenes when a new pod is extracted by the queue of pods to be scheduled (Fig. 3). During this phase, each node is assigned a score that reflects its suitability for hosting the replica, where the node having the highest score is considered for task scheduling. This suitability is assessed based on evaluations performed within the Custom Plugin. Specifically, it requests scheduling suggestions from the RL model via Suggestions APIs, exposed by a server that acts as an intermediary, facilitating seamless communication and ensuring that the scheduler can access up-to-date recommendations when making scheduling decisions. RL-ICE leverages the information provided by Prometheus, a monitoring system that exposes nodes performance indicators such as CPU usage, memory consumption, and network metrics, to select the best node, to which the custom Plugin will assign a higher score. The integration of RL within the scheduler is a key innovation, as it brings adaptive learning capabilities to the scheduling process. By continuously learning from the environment and task execution outcomes, RL-ICE can dynamically adjust its scoring criteria, leading to progressively better scheduling decisions over time.

The user can express its intents in the deployment request YAML file and, more specifically, in the *spec:template:metadata:annotations* section of the deployment request. The system attempts to satisfy this request, and to measure user-perceived latency values we deploy a dedicated service in each cluster that periodically runs *ping* commands. Given that a cluster consists of multiple nodes located geographically close to one another, particularly in edge computing environments, we assume minimal latency variability within the cluster. Initially, the latency service measures the latency between one node and the user. Once pods are deployed, those nodes will be used to ping the user and gather actual latency measurements for further requests.

TABLE I: Settings for experiments with varying cardinality.

# Clusters	# Nodes	# Pod requests
5	50	100
5	100	250
5	250	850
5	500	1500

TABLE II: Types of nodes in the continuum as from the EC2 on-demand instances in US West N. California.

Instance name	vCPU	Memory (GiB)	Bandwidth (Gbps)	Price $/hour
t3a.small	2	2	5	0.0223
t4g.xlarge	4	16	5	0.16
m7gd.2xlarge	8	32	15	0.5027
r5n.4xlarge	16	128	25	1.352

VI. Evaluation

In this section, we evaluate the advantages of RL-ICE across two testbeds: first, through a simulation to assess its scalability and, later, in a real Kubernetes environment.

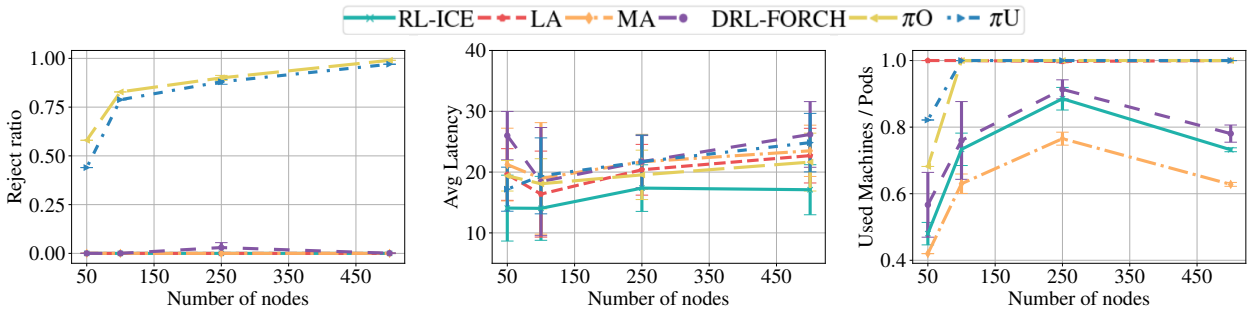
A. Experimental Settings

We train the model in a setting with 5 clusters and 10 nodes each, although tests consider a number of nodes in each cluster varying between 10 and 100 as described in Table I. For the higher level model, the train is composed of 100000 steps, 0.0004 learning rate, 0.96 gamma, $\alpha = 1$. After each action (cluster selection), the node is selected based on the highest total fractional resource margin after the pod placement. For the low level model, the training is run for 200000 steps, with 0.00065 learning rate, 0.967 gamma and $\beta = 0.02$. For both models we adopt a target network frequency of 500.The training is divided in episodes with different nodes and pods requirements to avoid over-fitting to one scenario.

Benchmarks. We compare our solution against these benchmarks:

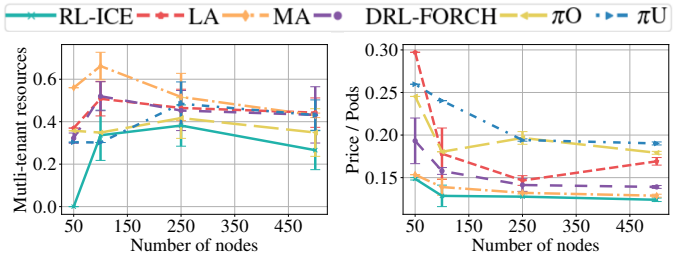
- Least Allocated (LA): the default scheduling policy in K8s. It selects the machine with the lowest resource utilization to perform load balancing.
- Most allocated (MA): an alternative strategy in the K8s Scheduler, selecting the machine with the highest resource utilization to reduce the number of physical machines in use.
- πO [14]: an RL strategy that assigns a reward of 1 for each successfully scheduled pod, aiming to maximize throughput.
- πU [14]: an RL formulation aimed at improving energy efficiency, where the reward is based on the average power consumption across all normalized resource utilizations of each machine.
- DRL-FORCH [16]: an RL orchestration policy whose input state captures each node's utilization and service requirements, thereby considering all available nodes to make informed placement decisions. The reward penalizes blocking and violations of latency service requirements.

Continuum Settings. In our simulation, we examine five clusters of nodes representing different tenants, each with a varying number of nodes in different experiments. These



(a) (b) (c)

Fig. 4: RL-ICE reduces user-perceived latency and energy consumption.



(a) Ratio of pods allocated to other tenants

(b) Price per pod

Fig. 5: RL-ICE prioritizes the resources of the tenant and allocates pods in a cost-effective manner.

clusters can be situated at either the close edge or the far edge, impacting the latency perceived by the users. The latency ranges from 5 to 10 ms for close edge clusters and from 15 to 50 ms for far edge clusters. The node types are defined by available resources (i.e., CPU, RAM, bandwidth) and pricing based on real EC2 instances [40] of the US West (N. California) region, detailed in Table II. Each cluster contains a mix of node types, randomly selected. Pod requests are characterized by random resource demands (but within defined ranges), as well as specific latency requirements and lifetime constraints. We conduct 35 tests with different seeds for each setting and plot the confidence intervals.

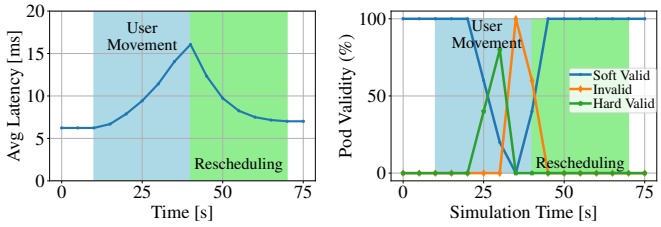
B. Experimental Results

We first compare RL-ICE to the benchmarks in terms of pods rejected, average latency, energy optimization and cost efficiency, while also examining their scalability as the number of nodes and pods increases in each experiment. Table I shows the setting for each experiment. We start showing the ratio of rejected requests in Fig. 4a, the average latency of pods in Fig. 4b, and the ratio of physical non-idle nodes divided by the number of allocated pods in Fig. 4c. The two policies πO and πU face many difficulties in placing the pods in nodes with enough resources. Since their input state is composed of utilization and capacity of all machines, along with pod resource requests at time *t*, it is clear that input size depends on the number of nodes and, for this reason, we needed to re-train the model for each setting. Despite this, since the state size explodes for large-scale settings, it doesn't learn to map the correct action to the input state, leading to poor throughput and energy efficiency. As such, they accept a very limited number of requests, and it happens that sometimes they manage to have a low latency, but they are actually agnostic to it. The default Kubernetes scheduling mechanisms effectively utilize node resources to accommodate requests; however, they fail to meet latency constraints. The Least Allocated (LA) policy distributes pods across all nodes, resulting in an average latency of approximately $25ms$ in all experiments, which reflects the average latency across all clusters. In contrast, the Most Allocated (MA) policy aims to saturate the resources of each node before moving on to the next, leading to a measured latency that is significantly influenced by the first nodes selected, in this case higher than LA. These policies serve as the best and worst baselines when tracking the number of used machines by each algorithm.

DRL-FORCH, similarly to RL-ICE, can generalize over large-scale settings without the need for retraining; however, even though it uses a reduced number of physical machines (Fig. 4c), it leads to average latencies that barely meet services latency requirements (Fig. 4b). On the contrary, RL-ICE shows a significant improvement in energy optimization and average pod latency. The performance gap between RL-ICE and the baseline MA policy stems from two reasons: *(i)* our scheduling mechanism seeks a balance between energy optimization and cost minimization, *(ii)* our hierarchical model's ability to select clusters based on latency requirements leads to the selection of unused nodes, even when active ones still have available resources. In contrast, the MA policy operates on a cluster-agnostic global set of nodes.

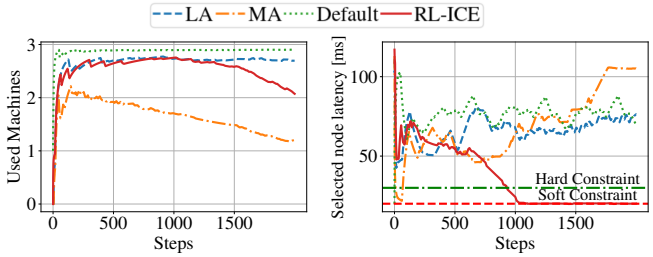
Fig. 5a keeps track of the percentage of pods allocated in other tenants resources, which implies an additional cost, while Fig.5b shows the cost of the deployment, computed as the sum of all the used nodes price, divided by the number of accepted pod replicas. Overall, Fig. 5 demonstrates that our model outperforms all the benchmarks in cost-effective decision-making.

For the next experiment we model three key events: Pod Arrival, Latency Measurement, and User Movement. The Pod Arrival rate follows an exponential distribution with a rate of 2 pods per second, once a pod's lifetime ends, it is descheduled. The Latency Measurement event collects current



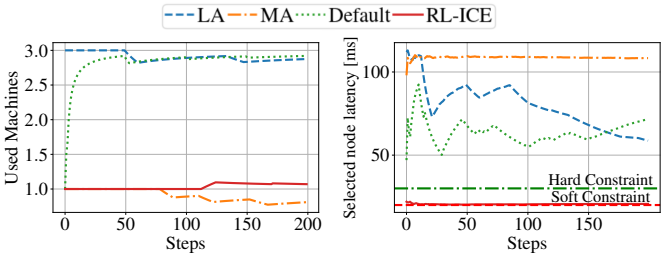
(a) Avg. latency over time (b) Pod validity over time

Fig. 6: When the user moves and, consequently, some pods become invalid, RL-ICE reschedule the replicas to satisfy their latency requirements.



(a) Energy efficiency (b) Latency

Fig. 7: Evolution and comparison of (a) number of nodes (b) user-perceived latency during the *training* of RL-ICE.



(a) Energy efficiency (b) Latency

Fig. 8: Evolution and comparison of (a) number of nodes (b) user-perceived latency during the *testing* of RL-ICE.

user-perceived latency and reschedules pods that are in invalid clusters when possible. This event occurs every 5s. The User Movement event simulates a scenario where the user moves to a different location or experiences a network disruption, resulting in increased latency towards a specific cluster. The latency variation lasts for $30s$ until the validity of pod placements is impacted and the rescheduling takes place.

In this experiment, 10 users are served, and 150 pod replicas are satisfied in total, we report the average latency in Fig. 6, showing how the model reacts to latency variations for a single user. Fig. 6a illustrates the variation of the average latency perceived by the user during the simulation, recorded at each *Latency Measurement Event*. Fig. 6b displays the percentage of Soft Valid, Hard Valid, and Invalid pods over time for the user, based on which latency requirements (soft, hard, none) are satisfied. At $10s$, the *User Movement Event* occurs (blue rectangle), gradually increasing the perceived latency (Fig. 6a). At first, from $25s$ to $30s$, the cluster latency rises above the Soft constraint but still below the Hard threshold, but, at around $35s$, the pod replicas all reach the Invalid status (Fig. 6b). The Latency Measurement registers the violation of the requirement at the subsequent measurement, at $40s$, and at this point, RL-ICE starts rescheduling the Invalid replica, looking for clusters that can satisfy the soft constraint. Once this is done, we can observe a reduction of the average latency and an increment of the Soft valid pods.

C. Kubernetes Prototype

We also prototyped our solution in a real Kubernetes Kind environment using *kindest/node:v1.27.3*, which runs on *Docker 20.10.23*. The plugin is programmed with *Go 1.21.4* and the RL model with *Python 3.8.10*. The cluster is composed of 3 nodes. In this experiment, we also test the default Kubernetes scheduler, which selects intra-cluster nodes in a round-robin fashion [41].

Fig. 7 shows the number of used machines and latency moving average during training for multiple policies. The Most Allocated (MA) policy acts as a baseline for energy efficiency, minimizing the number of used machines. However, as the Default scheduler and LA policy, MA is agnostic to latency measurements. On the contrary, RL-ICE progressively learns, through trial and error, to select the node that meets the latency requirements specified by the user (b). Once the exploration phase is done in (a) at around 1000 steps, RL-ICE starts exploiting the acquired knowledge, condensing the pod replicas in a lower number of machines, and moving closer to the baseline policy. This behavior confirms the ability of both RL models in our solution to effectively learn the policy.

Indeed, during the testing (Fig. 8), the number of used machines is always at its minimum and similar to MA (Fig. 8a), showing evident improvement in energy consumption w.r.t. LA policy and the Default scheduler. Similarly, the pod latency always satisfies both user-defined constraints, unlike other schema (Fig. 8b).

VII. Conclusion

In this paper, we propose RL-ICE, a hierarchical reinforcement learning-based scheduler for cloud-edge continuum that balances user latency requirements, cost optimization, and energy efficiency by selecting the optimal cluster and node. By incorporating DeepSets, RL-ICE efficiently handles large-scale scenarios and adapts to failures. Our evaluation in both simulated and real Kubernetes environments shows that RL-ICE outperforms state-of-the-art schedulers in balancing multiple objectives, minimizing latency, and reducing resource usage. Additionally, it dynamically responds to user movement and network failures by migrating microservices as needed. In future works, we will take into consideration the interdependence among pods and evaluate the computational overhead of our solution.

Acknowledgment

This work has received funding from EU Horizon Europe R&I Programme under Grant Agreement no. 101070473 (FLUIDOS).

REFERENCES

[1] O. Bentaleb, A. S. Belloum, A. Sebaa, and A. El-Maouhab, "Containerization technologies: Taxonomies, applications and challenges," *The Journal of Supercomputing*, vol. 78, no. 1, pp. 1144–1181, 2022.

[2] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 17, p. e5668, 2020.

[3] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. 7, pp. 152 443–152 472, 2019.

[4] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–37, 2022.

[5] G. Carvalho, B. Cabral, V. Pereira, and J. Bernardino, "Edge computing: current trends, research challenges and future directions," *Computing*, vol. 103, no. 5, pp. 993–1023, 2021.

[6] J. Singh, P. Singh, and S. S. Gill, "Fog computing: A taxonomy, systematic review, current trends and research challenges," *Journal of Parallel and Distributed Computing*, vol. 157, pp. 56–85, 2021.

[7] A. Sacco, F. Esposito, P. Okorie, and G. Marchetto, "LiveMicro: An Edge Computing System for Collaborative Telepathology," in *Proceedings of the 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge '19)*. USENIX, 2019.

[8] O. Tomarchio, D. Calcaterra, and G. D. Modica, "Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks," *Journal of Cloud Computing*, vol. 9, no. 1, p. 49, 2020.

[9] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi, "A unified model for the mobile-edge-cloud continuum," *ACM Transactions on Internet Technology (TOIT)*, vol. 19, no. 2, pp. 1–21, 2019.

[10] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2021.

[11] C. Chiaro, D. Monaco, A. Sacco, C. Casetti, and G. Marchetto, "Latency-aware Scheduling in the Cloud-Edge Continuum," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. IEEE, 2024, pp. 1–5.

[12] J. Huang, C. Xiao, and W. Wu, "Rlsk: A job scheduler for federated kubernetes clusters based on reinforcement learning," in *2020 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2020, pp. 116–123.

[13] Y. Zhang, B. Di, Z. Zheng, J. Lin, and L. Song, "Distributed multi-cloud multi-access edge computing by multi-agent reinforcement learning," *IEEE Transactions on Wireless Communications*, vol. 20, no. 4, pp. 2565–2578, 2020.

[14] J. Rothman and J. Chamanara, "An RL-Based Model for Optimized Kubernetes Scheduling," in *2023 IEEE 31st International Conference on Network Protocols (ICNP)*. IEEE, 2023, pp. 1–6.

[15] 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.

[16] 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.

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

[18] 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.

[19] N. D. Nguyen, L.-A. Phan, D.-H. Park, S. Kim, and T. Kim, "Elasticfog: Elastic resource provisioning in container-based fog computing," *IEEE Access*, vol. 8, pp. 183 879–183 890, 2020.

[20] Ł. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–9.

[21] A. Marchese and O. Tomarchio, "Network-aware container placement in cloud-edge kubernetes clusters," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 859–865.

[22] H. Li, H. Liu, C. Liu, A. Chen, Z. Niu, and J. Du, "Neilats: Neighbor-aware latency-sensitive application scheduling in heterogeneous cloud-edge environment," in *Proceedings of the 52nd International Conference on Parallel Processing*, 2023, pp. 615–624.

[23] 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*, vol. 20, no. 4, pp. 4461–4477, 2023.

[24] K. Govindarajan, C. Govindarajan, and M. Verma, "Network aware container orchestration for telco workloads," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE, 2022, pp. 397–406.

[25] M. Sun, S. Quan, X. Wang, and Z. Huang, "Latency-aware scheduling for data-oriented service requests in collaborative iot-edge-cloud networks," *Future Generation Computer Systems*, p. 107538, 2024.

[26] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.

[27] F. Jiang, K. Ferriter, and C. Castillo, "A cloud-agnostic framework to enable cost-aware scheduling of applications in a multi-cloud environment," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[28] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, "mck8s: An orchestration platform for geo-distributed multi-cluster environments," in *2021 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2021, pp. 1–10.

[29] A. Sacco, F. Esposito, and G. Marchetto, "Restoring application traffic of latency-sensitive networked systems using adversarial autoencoders," *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 2521–2535, 2022.

[30] W.-K. Lai, Y.-C. Wang, and S.-C. Wei, "Delay-aware container scheduling in kubernetes," *IEEE Internet of Things Journal*, vol. 10, no. 13, pp. 11 813–11 824, 2023.

[31] C. Centofanti, W. Tiberti, A. Marotta, F. Graziosi, and D. Cassioli, "Latency-aware kubernetes scheduling for microservices orchestration at the edge," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. IEEE, 2023, pp. 426–431.

[32] D. Milojicic, "The edge-to-cloud continuum," *Computer*, vol. 53, no. 11, pp. 16–25, 2020.

[33] S. Galantino, E. Albanese, N. Asadov, S. Braghin, F. Cappa *et al.*, "Building the Cloud Continuum with REAR," in *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*. IEEE, 2024, pp. 67–72.

[34] "Horizontal Pod Autoscaling," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, Accessed: January 24, 2025.

[35] L. Vu, T. Vu, T.-L. Vu, and A. Srivastava, "Safe exploration reinforcement learning for load restoration using invalid action masking," in *2023 IEEE Power & Energy Society General Meeting (PESGM)*. IEEE, 2023, pp. 1–5.

[36] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser *et al.*, "Starcraft ii: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[37] S. Huang and S. Ontañón, "A closer look at invalid action masking in policy gradient algorithms," *arXiv preprint arXiv:2006.14171*, 2020.

[38] A. Sacco, M. Flocco, F. Esposito, and G. Marchetto, "Owl: Congestion control with partially invisible networks via reinforcement learning," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[39] M. Iorio, F. Risso, A. Palesandro, L. Camiciotti, and A. Manzalini, "Computing without borders: The way towards liquid computing," *IEEE Transactions on Cloud Computing*, vol. 11, no. 3, pp. 2820–2838, 2022.

[40] "Amazon EC2 On-Demand Pricing," https://aws.amazon.com/it/ec2/pricing/on-demand/, Accessed: September 2, 2024.

[41] "Scheduler Performance Tuning," https://kubernetes.io/docs/concepts/scheduling-eviction/scheduler-perf-tuning/, Accessed: April 17, 2024.