Runtime Management of Artificial Intelligence Applications Through Hierarchical Reinforcement Learning

Riccardo Cavadini[1][0000−0002−7581−5955], Hamta Sedghani[1][0000−0002−6495−5717], Federica Filippini[1][0000−0002−6549−924X], and Danilo Ardagna[1][0000−0002−6549−924X]

Politecnico di Milano, 20133 Milano, Italy {name.surname}@polimi.it

Abstract. New and evolving Artificial Intelligence (AI) applications span across the full spectrum of computing resources, seamlessly integrating across Edge and Cloud platforms. Deploying applications at the network edge reduces latency, while cloud computing ensures a higher processing power. Managing resources in such a dynamic and diverse environment requires a strategic approach to satisfy Quality of Service requirements and minimize costs. To address this challenge, we propose FIGARO (reinForcement learnInG mAnagement acRoss computing cOntinuum), which exploits offline training and imitation learning to speed up the training of reinforcement learning-based agents able to control resources in the full cloud continuum stack. By extending our framework, we designed a hierarchical system structure and tested agents that only need to manage one computational layer at a time. This approach enables the system to efficiently manage multiple application components in complex AI pipelines. The results demonstrate the effectiveness of the hierarchical method, as the local agents dynamically scale computational resources, limiting QoS constraint violations to a maximum of 1.4% in the reference use-case application.

Keywords: Reinforcement Learning · Computing Continuum · Artificial Intelligence

1 Introduction

The amount of data that needs to be processed per time unit has drastically increased over the last few years, particularly with Artificial Intelligence (AI) applications. This trend is not expected to change in the future, with the AI market projected to grow more than 6x by 2030, driven by a compound annual growth rate (CAGR) of 35.7% [2]. The computational landscape witnessed a huge paradigm shift, moving from applications executed in pure cloud environments to a *computing continuum* of diverse resources with largely heterogeneous characteristics, ranging from small Internet of Things (IoT) devices to large data centers. On such an integrated system, applications workflows can run at

the periphery of the network, supported by a very low latency but limited computational capabilities, or they can be offloaded to remote services, e.g., cloud Virtual Machines (VMs) or Functions as a Service (FaaS) instances.

A well-known problem in this field is the Resource Selection and application Component Placement problem (RS-CP). Mapping the workload entering the different components of an application pipeline is fundamental to appropriately support their execution; however, choosing which resource to devote to run each part of the pipeline gets harder as the complexity of the application workflow increases. The RS-CP problem is usually tackled at design time (e.g., in [23, 26]) to determine the minimum-cost solution that satisfies Quality of Service (QoS) constraints. While design-time tools are usually effective in identifying the proper resource types to be deployed in the system and an initial placement of application components on the available computational layers, they usually cannot compute the optimal solution in real-time. Moreover, and most importantly, any change in the system status (e.g., a demand increase or unexpected workload fluctuations) would result in performance degradations or in a significant impact on system costs if the design-time solution were not adapted accordingly.

We propose FIGARO (reinForcement learnInG mAnagement acRoss the computing cOntinuum) to address the periodic adaptation of the RS-CP solution at runtime, leveraging Deep Reinforcement Learning (DRL) to automatically learn an effective management policy under varying environmental conditions. To address one of the most prominent issues in the development of RL algorithms, i.e., the huge time required to learn a good enough policy, FIGARO leverages the design-time knowledge to guide the agent decisions in the early stages of the training process. The policy is then refined and continuously adapted through interactions with an evolving environment.

In this paper, we extend our previous work [8], which was characterized by significant limitations and scalability problems. In fact, the RL agent we developed could control only systems involving few application components due to the increasing complexity of the underlying environment and agent state. We envision a hierarchical framework where multiple *local* agents are deployed on different computational layers. Each of them will be responsible for a single resource hosting (possibly) multiple co-located components, and it will dynamically scale the number of resource instances to limit the execution costs while satisfying the user-imposed QoS constraints. A centralized *master* agent with a global view of the computing continuum environment and the components pipeline will then support the overall execution by taking higher-level decisions involving, e.g., the migration of components from one resource to another.

In summary, our main contributions are the following:

- We propose a novel hierarchical RL setting, which allows our FIGARO framework to easily scale to support complex application pipelines; moreover, we propose a Deep Q-Learning-based implementation for the local agents, effectively designing the observations of the agent to support the execution of multiple co-located components.

– We boost the ability of the agent to learn an optimal policy by leveraging Behavioral Cloning (BC) [21] to exploit the experience collected by SPACE4AI-D, a state-of-the-art design-time tool for AI applications capacity planning in the computing continuum [23];
– We validate our work by conducting several experiments considering multi-component application pipelines deployed on edge and cloud resources, including a reference use-case application that supports Deep Neural Network (DNN) partitioning. The results prove the effectiveness of our hierarchical approach, since local agents manage to dynamically scale the computational resources, reaching at most 1.4% of QoS constraints violations at the end of the training in the reference use-case application.

The rest of the paper is organized as follows: Section 2 describes the RS-CP problem at runtime. Section 3 presents a two-component AI application that we use as a reference in our work. Section 4 offers a complete overview of FIGARO. The experimental validation of the framework is based on numerical analyses and is discussed in Section 5. Section 6 briefly overviews related works. Conclusions are finally drawn in Section 7.

2 RS-CP Problem

The general goal of the RS-CP problem is to minimize the execution costs of computing continuum resources while complying with memory constraints, components to resources compatibility, and QoS requirements.

Some works in literature [23] propose ways to solve the problem at design time. This means that, given all the computational layers of the computing continuum, decisions are made on:

1. which resources to utilize at each layer of the computing continuum;
2. which layer to use for executing various application components.

At runtime, the design-time solution is considered a starting point, and the deployment is re-evaluated in the new system conditions.

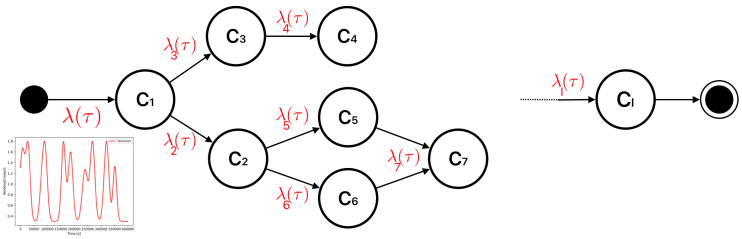


Fig. 1: Application DAG.

Following the approach described in [23] for the design-time scenario, we model AI inference applications as Directed Acyclic Graphs (DAGs). Each node

represents a different application component (see Figure 1), denoted by set $\mathcal{I}$. We assume each component $i \in \mathcal{I}$ to be the inference step of a DNN running in a software container. This can, in principle, be deployed on heterogeneous resources, including edge devices (denoted by the set $\mathcal{J}_{\mathcal{E}}$) and cloud VMs (denoted by $\mathcal{J}_{\mathcal{C}}$). For the sake of simplicity, we consider DAGs with a single entry point, receiving as input an exogenous workload $\lambda(\tau)$ (expressed in requests per second) for every time interval τ, and a single exit point. DAG edges represent the data transfer between application components, therefore including information about data size and transition probability. DNNs might be split so that different partitions (e.g., groups of convolutional layers) are executed in a distributed manner. A given DNN may be partitioned differently according to resource capacity and network settings. Therefore, each component is characterised by a set $\mathcal{C}^i$ of candidate deployments, such that $c_s^i \in \mathcal{C}^i$ is a set of neural network partitions, i.e., $c_s^i = \{\pi_h^i\}_{h \in \mathcal{H}_s^i}$.

We denote by R_i the response time of component $i \in \mathcal{I}$, defined as the sum between the execution time $\tilde{R}_h^i$ of all partitions in the selected deployment and the expected delay due to data transmissions between them. Similarly, if we denote by P a *path* including a sequence of consecutive components, we can define its response time $\hat{R}_P$ as the sum of R_i for all components in P, plus the transmission delays between different components. Accordingly, QoS requirements may be imposed in the form of *local constraints*, i.e., thresholds $\overline{LR}_i$ on the maximum response times of single components, or *global constraints*, i.e., upper bounds $\overline{GR}_P$ on the execution times of multi-component paths.

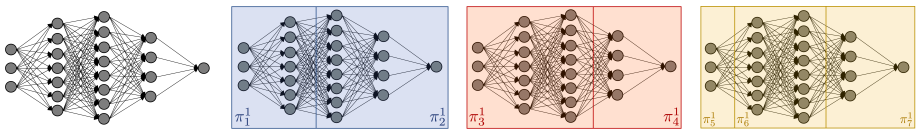


Fig. 2: Example of AI application component with its candidate deployments

In our work, as in [23], the response times of all components and partitions are computed by exploiting M/M/1 queues, considering the service demands D_i. This means that inter-arrival and service times follow an exponential distribution. This assumption does not hold at runtime (in our previous work [8], we relied on a simulator that allows us to consider heterogeneous service times distributions for validation purposes). At design time, a computational layer is a set of resources that can be chosen as an alternative to each other. At runtime, which is our case, the resource to use has already been selected. Therefore, we can define the computational layer via the specific kind of computational resource (i.e., VM) that it contains, with its parameters such as memory, cost, and number of cores. Each kind of VM can be instantiated multiple times, from a minimum of 1 (or 2 for reliability reasons) to a maximum number N.

Furthermore, executing consecutive components or partitions on different resources requires data transfer among the selected devices. Communications are enabled by different network domains, exploiting heterogeneous technologies in terms of access delays and available bandwidth (e.g., WiFi, 5G).

Edge devices and cloud VMs are characterized by hourly costs, denoted by C_E, and C_C, respectively, which account for resources and energy consumption.

The problem of finding a component placement that minimizes the total execution costs while satisfying hardware, network, and QoS constraints can be formulated as a Mixed Integer Non-Linear Program (MINLP) with the following objective function (see [23]):

$$\min C_E + C_C. \tag{P1a}$$

s.t.

$$R_i \leq \overline{LR}_i \qquad \forall (i, \overline{LR}_i) \in \mathcal{LC} \tag{P1b}$$

$$\hat{R}_P \leq \overline{GR}_P \qquad \forall (P, \overline{GR}_P) \in \mathcal{GC}, \tag{P1c}$$

where $\mathcal{LC}$ and $\mathcal{GC}$ are the sets of components or execution paths included in local or global constraints, respectively. Additionally, the optimization is subject to constraints related to the memory availability in different resources and the assignment compatibility between components and resources. Solving this problem at runtime means switching on/off the appropriate edge resources, scaling in/out the cloud VM instances, identifying a deployment for each component, allocating the partitions on the chosen devices, and checking compatibility with memory constraints and QoS requirements. The choice of the DNN deployment for each component is managed through the binary vector $\mathbf{Z}$, whose element z_s^i is 1 if c_s^i is selected for component $i \in \mathcal{I}$. Similarly, the component-to-resource assignments are described through the matrix $\hat{\mathbf{Y}}$. Each element $\hat{y}_{hj}^i$ is equal to the number of instances of resource $j \in \mathcal{J}$ assigned to partition π_h^i.

To effectively tackle the RS-CP problem in a dynamic runtime setting, this MINLP can be reformulated as a Markov Decision Process and then addressed with an RL-based approach as will be presented in Section 4.

3 Sample Use-Case Application

As a reference case study, we consider a two-component AI application (see Figure 3) initially proposed in [20]. The first component, denoted as *blurry-faces*, receives as input a video, recorded, e.g., by security cameras in a public setting; it exploits the *FFmpeg* tool to extract a frame every 5 seconds and the YOLO (You Only Look Once) network [19] to detect faces appearing in it. These are then blurred to anonymize the frames, preparing them for subsequent processing. The second component, *mask-detector*, works on the images provided by *blurry-faces*; it leverages the YOLO network to identify faces in the pictures and classify them according to whether people are wearing a face mask.

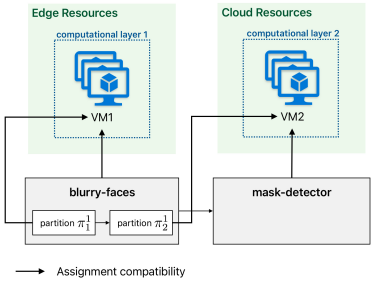


Fig. 3: Mask detection application.

As reported in the picture, the first component is characterized by two candidate deployments: the whole component can be executed exploiting an on-premises VM at the edge ($VM1$), or it can be partitioned so that π_2^1 is offloaded to $VM2$ in the cloud. Both solutions ensure that no sensitive data leaves the local resources, thus implicitly satisfying privacy constraints. However, delegating part of the processing to a cloud VM would significantly help in controlling the component execution time if the incoming load increases, saturating the limited computational capacity at the edge.

Both $VM1$ and $VM2$ are characterized by a maximum number of available instances; at design time, the tool in [23] identifies an initial production deployment according to an expected peak value of input workload and to the QoS constraints imposed on the two components response time. In particular, we may impose a threshold on the response time of each component (local constraint) or on the whole application execution (global constraint).

At runtime, FIGARO is responsible for adapting the production deployment to the current workload and conditions of the environment. As it was mentioned in Section 1 and will be clarified in the next sections, a *local* agent deployed on each computational layer will identify the most suitable number of instances for each running VM, leaving to a centralized *master* agent to determine which candidate deployment to select for the first component.

It is crucial to observe that the master agent decisions have a significant impact on the two local agents: in particular, the one deployed on the second computational layer should be able to generalize and take appropriate actions independently on the number of components (or component partitions) that are deployed on $VM2$. Similarly, the agent on the first computational layer faces different scenarios according to whether *blurry-faces* is partitioned or not, since the service demand and the response time constraint associated with the whole component are significantly different from those related to π_1^1.

4 FIGARO

FIGARO is a framework for the runtime management of AI applications. It is based on Reinforcement Learning, which makes it capable of adapting to the

dynamic nature of real-world environments. A relevant source of variability is the input workload, which can fluctuate over time for various reasons. This makes static resource assignment ineffective, as it can lead to either underutilization or saturation. FIGARO includes an offline learning process, followed by the deployment of the pre-trained agent in a real-world scenario. In this work, we focus only on the offline training phase of the agent, enhancing both the quality of the solution and the time required to achieve it.

4.1 Reinforcement Learning

RL algorithms include an *agent* that interacts with an *environment* to maximize a reward function. In each time window τ, the agent observes the current state $s(\tau)$ of the environment and selects an action $a(\tau)$ to take. Based on the outcome of this action, it receives a numeric reward signal $r(\tau)$.

In our case, as the main goal is to minimize the operational cost of the system, we include it in the calculation of reward for the agent. A detailed analysis of the reward computation is carried out in Subsection 4.5. The learning process involves continuous interaction with the environment, which transitions from state $s(\tau)$ to $s' = s(\tau')$ in response to the actions of the agent. The cost associated with each state-action pair depends on this transition (i.e., $c = c(s, a, s')$). The strategy followed by the agent to choose actions based on the current state is referred to as *policy*. In the following, we describe:

- the *state space*, representing all possible states the agent may encounter;
- the *action space*, representing the set of all possible actions the agent can choose from;
- the *cost*, which depends on the selected action and the next observed state.

The RS-CP problem described in Section 2 faces a rapidly growing complexity. Specifically, the sizes of the assignment matrix $\hat{\mathbf{Y}}$ and the deployment vector $\mathbf{Z}$ grow with the number of layers and components, while the size of the response times vector $\mathbf{R}$ grows with the number of components. As a result, the state space can easily become significantly large, making it challenging for the agent to converge to an optimal policy.

To address this problem, we utilized the *Deep Q-Network* (DQN) algorithm, which effectively combines RL and DNNs [8]. DQN is based on the concept of Q-function, which evaluates the expected rewards of taking a specific action a in a given state s and then following policy π. In cases where there are many combinations of states and actions, such as ours, a DNN is used to approximate the Q-function.

During training, at each step, the experience of the agent is saved in an *experience replay buffer*. The saved tuple is: $(s(\tau), a(\tau), r(\tau), s(\tau + 1))$. By randomly sampling batches of experience tuples from this buffer during training, we can prevent the agent from becoming biased by correlations between consecutive events, improving both learning stability and efficiency [10].

DQN employs an epsilon-greedy strategy to balance exploration and exploitation during training. With probability ε, the agent selects a random action (exploration), and with probability $1-\varepsilon$, it selects the action that maximizes the Q-value (exploitation). As training progresses, ε is gradually reduced, allowing the agent to progressively rely more on the learned Q-function for decision-making.

4.2 Hierarchical architecture

To address the issue of *state explosion* [27], we adopted a hierarchical approach. The system is divided into sub-systems, each corresponding to a computational layer. The objective is to train as many agents as the number of sub-systems, with each agent controlling a limited number of local components that run on its respective computational layer. This greatly simplifies the state space for each agent. This design ensures that the policies of the agents can converge to optimal solutions, as demonstrated by our previous implementation [8]. However, the main challenge of this approach is enforcing cross-layer global constraints, as agents are unaware of other computational layers.

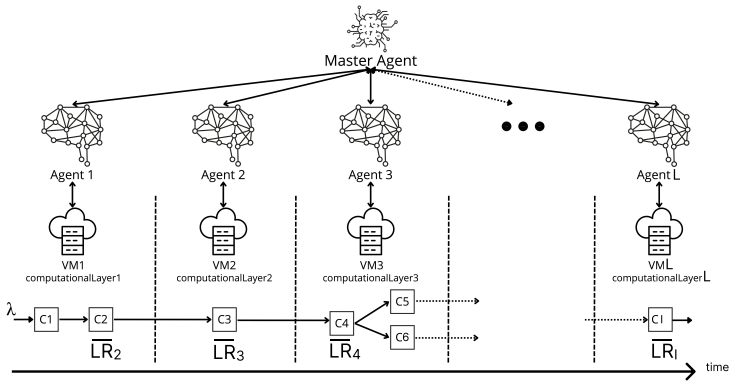


Fig. 4: Representation of the proposed layered architecture with a *master* agent that collects information about all the agents on the different computational layers, other than managing local and global constraints. Each agent is unaware of the others: they all make independent decisions related to their assigned sub-system. Each sub-system includes one computational layer with one VM and the components allocated to its layer. λ is the exogenous workload that enters the system and flows through the DAG.

We implemented the hierarchical system depicted in Figure 4. While each local agent manages a single computational layer, we introduced a higher-level *master* agent to coordinate them. Its roles include partitioning components, migrating them between layers, and splitting global constraints into local constraints and, possibly, local into further sub-level constraints.

Each sub-system, managed by a single agent, consists of one computational layer with a specific resource type (multiple instances of which can be selected),

and of the components allocated to that layer. This setup facilitates the handling of local constraints, represented by $\overline{LR}_i$, assigned to components within each computational layer.

To illustrate the role of agents, we discuss the use case presented in Section 3. As said, each local agent is responsible for scaling (up or down) VM instances of a specific type based on the input workload to that layer. The master agent oversees the entire system, making higher-level decisions at each time step. Its tasks include partitioning *blurry-faces*, if needed, and assigning components and partitions to compatible layers. Initially, components are assigned to computational layers according to the design-time configuration. Currently, the master agent relies on SPACE4AI-D [23] for component migration. Future work will include training a higher-level policy as well, specifically for this task.

Once a component or partition is assigned to a computational layer, the master agent handles both global and local constraints using its system-wide knowledge. It converts global constraints into local ones and assigns them to the appropriate components. For example, in the case depicted in Figure 3, a global constraint $\overline{GR}_1$ imposed on the entire application cannot be effectively managed by the two local agents, which have visibility only on the specific components deployed on $VM1$ or $VM2$. The local agent on the first layer, for instance, would not be able to scale the $VM1$ instances properly without knowing how close the system is to exceeding the threshold. This is because the overall response time also depends on *mask-detection*, and therefore on the decisions made by the agent on $VM2$. To address this, the master agent converts $\overline{GR}_1$ into local thresholds for the two components, assigning them in proportion to their respective service demands.

The same approach is applied when a local constraint $\overline{LR}_1$ is imposed on *blurry-faces* and the component is partitioned, since π_1^1 and π_2^1 are deployed on different resources (and thus controlled by different agents).

Algorithm 1 outlines the method applied to split global constraints. As defined in Section 2, let $\mathcal{GC}$ and $\mathcal{LC}$ be the set of global and local constraints, and D_i be the service demand of each component i. Looping over the paths P (and corresponding thresholds $\overline{GR}_P$) in $\mathcal{GC}$, we compute the proportion of $\overline{GR}_P$ that should be assigned to component $i \in P$ by considering the ratio between D_i and the overall demand of the path (see lines 3 – 6). The local threshold $\overline{LR}_i$ is then updated by considering the minimum over the contributions in $\overline{GR}_P^i$, to guarantee that the most restrictive constraint is considered, and by comparing this with the original upper bound if the component i was already involved in a local constraint (see lines 8 – 12).

With the previous approach, the system was able to manage only two computational layers hosting single components. The result of implementing the described hierarchical system is that each agent can manage complex pipelines, including multiple components, either co-located in a computational layer or deployed on different layers in the computing continuum.

Algorithm 1 Global constraints split per computational layer

Require: $\mathcal{GC}$, $\mathcal{LC}$, D_i $\forall i \in \mathcal{I}$

$\overline{GR}_P^i = \emptyset$ $\forall i \in P$, $\forall P \in \mathcal{GC}$
for each $(P, \overline{GR}_P) \in \mathcal{GC}$ **do**
  $D_P = \sum_{i \in P} D_i$ ▷ Sum the demands of the components i in the path
  for each $i \in P$ **do**
    $\overline{GR}_P^i \leftarrow \overline{GR}_P^i \cup \left\{ \frac{D_i}{D_P} \overline{GR}_P \right\}$ ▷ Compute the proportional local constraint
  end for
end for
if $i \in \mathcal{LC}$ **then** ▷ Assign the most restrictive constraint to the component i
  $\overline{LR}^i \leftarrow \min \left\{ \overline{LR}^i, \min_{\substack{\rho \in \overline{GR}_P^i \\ P \in \mathcal{GC}}} \rho \right\}$
else
  $\overline{LR}^i \leftarrow \min_{\substack{\rho \in \overline{GR}_P^i \\ P \in \mathcal{GC}}} \rho$
end if

4.3 Offline Pre-Training

Before entering into the Reinforcement Learning realm, we implemented a Behavioral Cloning (BC) process. In particular, we pre-trained agents by providing them with the behavior of SPACE4AI-D [23]. This tool, fed by the current workload, can identify the minimum cost deployment that fulfills QoS constraints for the set of considered components. In the long run, this allows us to limit violations and avoid unexpected and random behaviors when deployed in production. We implemented this by filling the replay buffer with previously obtained experiences from SPACE4AI-D. Over time, the agent interacts with the environment and collects new experiences, replacing the contents of the pre-filled replay buffer.

4.4 Local agents State Space and Actions

Each agent manages a single computational layer with a specified number of instances (all of the same type). The incoming workload is divided into local workloads for each agent. As illustrated in Figure 1, $\lambda(\tau)$ enters component $i = 1$: the output is then divided into $\lambda_2(\tau)$ and $\lambda_3(\tau)$ depending on the transition probabilities to components 2 and 3, respectively. Agents observe the incoming workload, and they should learn how to handle its variability. Under M/M/1 assumption, the response time of component i is computed as $R_i = \frac{D_i}{1-U}$, where D_i is the service demand of the component and U is the resource utilization in the computational layer to which the component is assigned. Therefore, each local agent sees a scalar value, U. To effectively manage the system, it is essential to identify the most critical component within each layer, i.e., the one closer to violating. To identify the critical component, we analyze the status of all components. We defined a *pressure* value as $\frac{R_i}{\overline{LR}_i}$, where $\overline{LR}_i$ is the response

time threshold of component i. We call *dominant* the component subject to the highest pressure, i.e., the closest to violating its constraint. We included pressure and queue length of the dominant component into the state since they depend on the response time. Therefore, the state space definition proposed in this work is:

$$s(\tau) = \langle n, \lambda, p, q, U \rangle \tag{1}$$

- n is the number of running instances of the resource in the layer, ranging from 1 to the maximum available number N;
- λ is the input workload;
- p is the pressure on the dominant component ($i = d$), computed as $\frac{R_d}{\overline{LR}_d}$, where R_d is the response time of the dominant component and $\overline{LR}_d$ is the associated threshold;
- q is the queue length of the dominant component, computed as $\frac{R_d - D_d}{D_d}$, where R_d is the response time of the dominant component and D_d is the demand of the dominant component (see [14]);
- U is the utilization of the layer.

Each agent can take scaling actions regarding the number of VM instances in its computational layer. The action space consists of integer values ranging from the minimum to the maximum number of VM instances available in the computational layer; at each timestep, the agent selects the number of instances to turn on.

4.5 Cost function

Through interactions with the environment, each local agent observes a cost, normalized between 0 and 1, which reflects the impact of its actions on various aspects of the environment state.

The primary factor we take into account is the violation of QoS constraints, which we consider as the most severe outcome. Therefore, we assign the maximum possible cost to any violation, as an agent should not try to reduce operational costs at the price of violating QoS requirements. When QoS constraints are met, the overall cost depends on the execution cost, calculated based on the selected instances and normalized against the maximum possible instance cost.

In Figure 5, we analyze the cost function using a sample workload within the range $[0, 10]$. In case an insufficient amount of VM instances is selected, resulting in a violation, the cost is set to 1. The optimal behavior would follow the bottom profile of the plot. In order to create a gap between the lower bound and the cost of violations, we use a parameter called β (set to 0.65 in the plot).

This significantly improves the cost function implemented in [8] and greatly increases the ability of the agent to discriminate between good and bad decisions while increasing the numerical stability of the training process.

Then, we add BC principles to influence also the cost calculation. In fact, as explained in Subsection 4.3, we want to exploit the existing knowledge from SPACE4AI-D at the beginning of the training. In order to do this, we set an

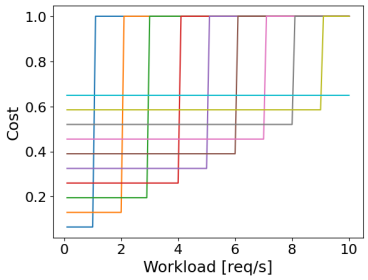


Fig. 5: Analysis of the cost function depending on varying workload ($\lambda \in [0, 10]$). Demand = 0.5, one VM type with a maximum of 10 instances that cost 2$/h each.

additional reward at the beginning of the training, given to the agent each time it chooses the same action that SPACE4AI-D would have chosen. We let this additional reward decay as time progresses, eventually reaching 0. This way, we avoid overfitting the behavior of SPACE4AI-D and allow the agent to learn the proper Q-value for the following iterations (i.e., the expected value of performing a scaling action given the current state).

So, the final reward function r for each local agent is defined as

$$1 - c + r^{BC}.$$

First, we define the cost as:

$$c(s,\ a,\ s') = \begin{cases} 1, & \text{if } \kappa^{LC}(s') = 1 \text{ or } \kappa^{GC}(s') = 1 \\ \frac{\kappa^{exec}(s,\ a,\ s')}{C^{max}} * \beta & \text{otherwise,} \end{cases} \tag{2}$$

where $\kappa^{LC}(s')$ and $\kappa^{GC}(s')$ are set to 1 if local or global constraints are violated, respectively, in state s'; κ^{exec} is the execution cost related to the chosen action (i.e., to the chosen number of VM instances), and C^{max} is the maximum execution cost. As already mentioned, β is a scaling term that represents the distance between the violation cost, and the maximum machine cost without any violation (see Figure 5).

Then, we define the additional reward for BC as:

$$r^{BC} = max(0, \alpha - ((m * \alpha)/M_{BC})), \tag{3}$$

where m is the index of the current step, and M_{BC} is the total number of steps for which we want to apply BC. α is the maximum reward gained through this process, which in our implementation is set to 0.5. r^{BC} linearly decays to 0 in M_{BC} iterations; then, the reward is evaluated just as $1 - c$.

Figure 5 shows the values of the cost function for different workload values after M_{BC} iterations. Agents should learn to minimize the cost, meaning that they will tend to follow the lower bound of the plot.

5 Experimental Results

This section validates the performance of FIGARO across various experiments using the Ray RLlib library [18]. To assess the effectiveness and scalability of FIGARO, we initially considered application pipelines of 4 components (see Section 5.1). Then, as discussed in Section 3, we tested our new implementation on a realistic pipeline with 2 components (see Section 5.2).

5.1 Evaluation of FIGARO

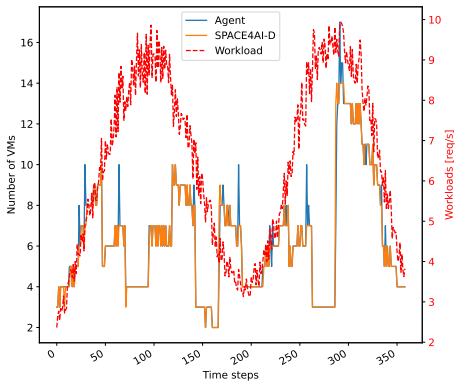
In this experiment, we consider application pipelines including up to 4 collocated components and conduct multiple trials, generating 5 random instances and reporting the results across the average of the 5 instances. We set a local constraint for each component as $10.1D_i$. The DQN algorithm employs an epsilon-greedy policy with both ε and the learning rate (lr) decaying linearly. As previously discussed, this allows for a balance between exploration and exploitation. Table 1 summarizes the parameter values used in the experiments. As shown in Table

Table 1: Simulation parameters

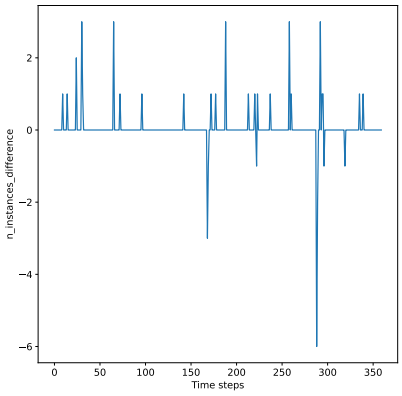
System Parameters	
N	20
I_l	4
λ	Bimodal trace in [0, 10] req/s
D_i	Uniform distribution in [0.1, 0.6] s
$\overline{R}_i$	$10.1 \times D_i$
c	2 $/h
Agent Parameters	
M	40000
Ω	$2^{I_l} - 1$
T	360 steps
lr	decaying from 0.01 to 0.001 during 30000 steps
ε	decaying from 0.2 to 0.01 during 30000 steps
Replay buffer capacity	360000
M_{BC}	30000
α	0.5
β	0.65
Network activation function	[ReLU, Tanh, ReLU]
Neuron per layer	[256, 256, 128]

1, each episode is 360 steps long, representing a 1-hour-simulation. In fact, the workload follows a bimodal distribution, with two peaks per episode: this is a realistic behavior observed in real systems [23]. Given that we assume at most four components can run on a computational layer ($I_l = 4$), the total number of possible scenarios (Ω) based on the components active on the layer is $2^4 - 1$. During training, the agent is trained over multiple episodes, with each episode focusing on a single scenario. In each subsequent episode, we switch to a different scenario. Conversely, during evaluation, the agent encounters all scenarios within

a single episode. Every T/Ω steps, we transition to a new scenario. It is important to note that scenarios are selected randomly during both the training and evaluation phases. We trained the agent for 40,000 steps, with evaluations conducted every 1,000 steps, each over a single episode. We used 10 rollout workers in RLlib and trained the network at every step. Before starting the training, we collected 360,000 samples of experience from SPACE4AI-D to pre-fill the replay buffer. Figure 6 presents a comparison between the agent and SPACE4AI-D during the final evaluation of a single instance. In Figure 6a, the blue line represents the performance of the agent, while the yellow line corresponds to the decisions made by SPACE4AI-D. The red dashed line shows the varying workload, which follows a bimodal distribution, fluctuating between peaks and valleys over time steps, with scenario switches occurring every 24 steps. The number of VMs on the y-axis reflects how both the agent and SPACE4AI-D allocate resources in response to the changing workload and scenario. Given that the goal was for the agent to closely mimic the behavior of SPACE4AI-D, the agent demonstrated a strong alignment, closely following the general trends in VM allocation in response to the component assignment scenario and workload variations. The alignment with the SPACE4AI-D resource management policy is particularly notable given the non-linearity of the workload and the complexity of the decision space. While some discrepancies (see Figure 6b), such as brief over-allocations of resources, are present, these deviations are relatively minor given the complexity of the task. The tendency of the agent to over-allocate resources during peak workloads and maximal component assignment (e.i. steps 288-312 in Figure 6) reflects its conservative approach to maintaining performance under high demand, a reasonable trade-off given the non-deterministic nature of the system. Further refinements can smooth out these fluctuations, but the current results already demonstrate robust performance within the challenging context of real-time, dynamic resource management.

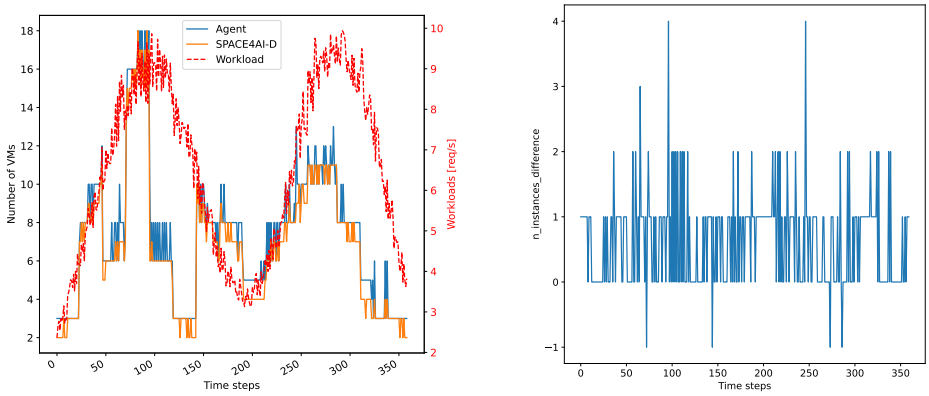


(a) Comparison of the agent's choices with those of SPACE4AI-D based on workload.



(b) Difference of the number of instances in use between the agent and SPACE4AI-D.

Fig. 6: Comparison between the agent and SPACE4AI-D, using BC.



(a) Comparison of the agent choices with those of SPACE4AI-D with varying workload.

(b) Difference of the number of instances in use between the agent and SPACE4AI-D.

Fig. 7: Comparison between the agent and SPACE4AI-D, without using BC.

To highlight the impact of BC, we ran the same random instance used in Figure 6 with identical experimental parameters but without employing BC. The results, presented in Figure 7 demonstrate the significant role of BC in solving the problem and improving the ability of the agent to mimic SPACE4AI-D. In the absence of BC, we observed a noticeable over-provisioning during evaluation (see Figure 7b).

Figure 8 illustrates the average values for key metrics—reward, violation, and absolute difference in the number of VMs—across 5 random instances using BC. These metrics were tracked during 40 evaluations interspersed throughout the training process. The reward and absolute difference in the number of VMs for a single evaluation are calculated as the average across all time steps within the evaluation episode (360 time steps). The violation percentage is calculated by dividing the total number of violations within an episode by the total number of time steps in that episode. These metrics values are then averaged across 5 random instances. The metrics demonstrate clear improvement over the course of training, showing that the agent is learning to both optimize resource allocation and minimize the violations.

The high fluctuations in violations and VM count differences at certain points can be attributed to the random selection of allocation scenarios in each random instance experiment, while the evaluation workload trace remains consistent across all instances. In some cases, the maximal component assignment scenario coincides with peak workload demand, leading to a higher number of violations and VM count differences in certain instances. This results in a higher average for these metrics compared to previous evaluations, causing noticeable spikes. In contrast, the jumps in the reward metric are smoother because the reward is normalized between 0 and 1.5, making it less sensitive to extreme fluctuations. As a result, even when maximal component assignment occurs during peak workloads, the impact on the reward is more gradual.

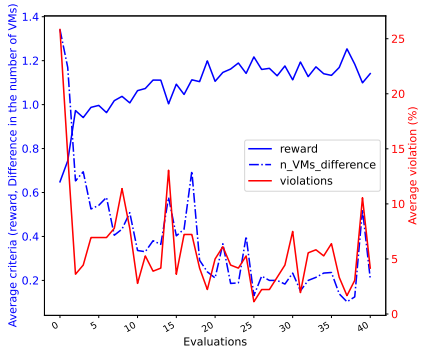
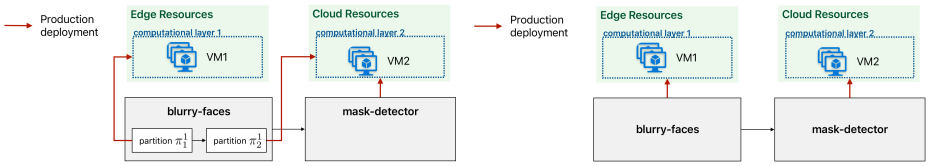


Fig. 8: Average reward, violation, and difference in the number of VMs, among 5 random instances.

5.2 Use-Case Application

In this section, we analyze the use case presented in Section 3 with two computational layers. As shown in Figure 3, each layer has two possible component assignment scenarios. For Layer 1, either the full *blurry-faces* component or the first partition of *blurry-faces* can be active at any given time. For Layer 2, either the *mask-detector* alone or both the *mask-detector* and the second partition of *blurry-faces* can be active simultaneously. Figure 9 illustrated these scenarios.

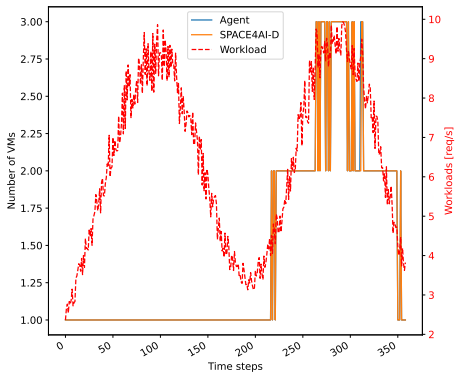


(a) Scenario 1. (b) Scenario 2.

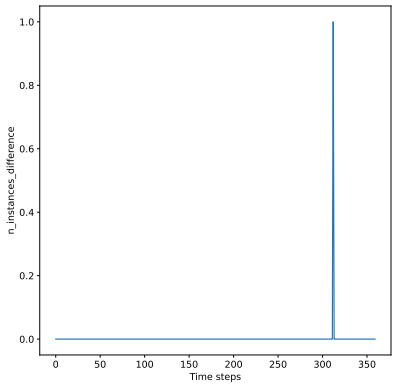
Fig. 9: Use-Case component assignment scenarios.

We set the demand of the full *blurry-faces* component and the *mask-detector* to 0.2 and 0.3, respectively, while the demand for the partitioned *blurry-faces* component is set to 0.05 for the first partition and 0.18 for the second. Additionally, we designated 2 and 3 seconds as the local constraints for the *blurry-faces* and *mask-detector* components, respectively. We trained the agent per layer for 8,000 steps, with the duration of the decaying epsilon and lr as well as N_{BC} all set to 7,000 steps. All other parameters remain as specified in Table 1.

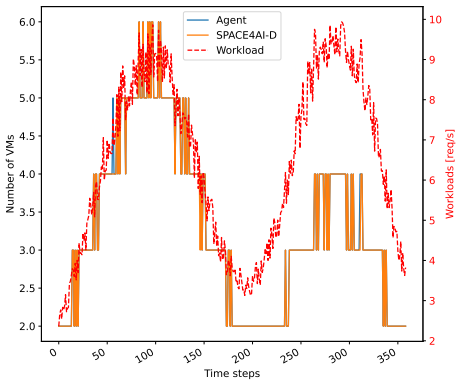
Figure 10 presents a comparison between the agents per layer and SPACE4AI-D during the evaluation phase. As previously discussed, each layer has two possible component assignment scenarios, and after half of the episode (180 time steps in Figures 10a and 10c), we switch from Scenario 1 (see Figure 9a) to Scenario



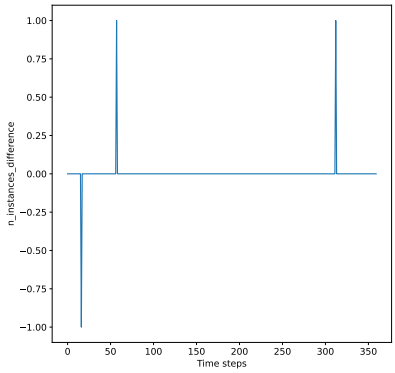
(a) Comparison of the agent choices with those of SPACE4AI-D, on $VM1$



(b) Difference of the number of instances on $VM1$



(c) Comparison of the agent choices with those of SPACE4AI-D, on $VM2$



(d) Difference of the number of instances, on $VM2$

Fig. 10: Comparison between the agent and SPACE4AI-D.

2 (see Figure 9b). In the first half of the episode, the partitioned deployment of the *blurry-faces* component is selected; thus, the first partition is running on $VM1$, and the second partition, along with the *mask-detector*, is running on $VM2$. This results in a high computational load on $VM2$, requiring 6 instances to meet the performance constraints, while $VM1$ experiences a light computational load, with only a single instance needed to satisfy its constraint. In the second half of the episode, the full *blurry-faces* component, which has higher resource demands, is selected, requiring more VM instances of $VM1$ to meet the performance constraints. Meanwhile, $VM2$ experiences a reduced load, as it is only running the *mask-detector* component. As shown, the agents successfully mimic SPACE4AI-D (small difference in 10b and 10d), dynamically allocating resources in response to the changing workload and scenario, thereby achieving a QoS constraint violation rate of 0% on $VM1$ and 1.4% on $VM2$.

6 Related Work

Recently, much research has been devoted to addressing the Resource Selection and Component Placement (RS-CP) problem as a continuous-control problem at runtime. Reinforcement Learning (RL) has increasingly been applied to the RS-CP problem in complex computing environments, spanning across edge, cloud, and mobile ecosystems. In fact, RL has been proven to be very helpful in solving complex multi-objective optimization problems related to highly dynamic environments.

[12] provides a comprehensive review of deep RL techniques to solve workflow scheduling problems in dynamic cloud and edge computing environments. The authors outline the most relevant challenges, such as multi-objectivity, the infamous curse of dimensionality, partial observability of the state, and multi-agent coordination.

For example, in [22], RL is used to manage the horizontal and vertical elasticity of container-based applications. Solving multi-resource allocation problems presents a critical challenge that can be effectively tackled with Deep Reinforcement Learning [28]. Authors have recently addressed it in multi-agent settings, where each RL agent can focus on a specific subtask, such as in [16]. In the present work, we follow the very same paradigm. In RL, when working with multiple agents, we can set them to cooperate or act independently from each other, depending on the design of the system and the addressed problem. Cooperative agents usually share information with each other with a shared objective, coordinating their actions to optimize the performance of the system. Instead, independent agents focus on optimizing their individual objectives: this behavior can lead to adversarial or conflicting strategies. Both paradigms have been shown to offer specific advantages, depending on the task to be performed and on the final objective. In the case of our setting, some solutions have demonstrated the efficacy of a hierarchical collaborative organization for the RS-CP problem [15].

Much research has been dedicated to optimizing resource usage and autoscaling in edge computing environments. A relevant contribution is the study by [24], which investigates serverless technologies in edge computing with RL algorithms. In particular, the study evaluates and compares various algorithms (DQN, A2C, and PPO) based on latency, estimation goodness, and generalization errors; PPO demonstrates superior convergence stability when facing demanding conditions in the environment. This leads us to consider PPO as a reasonable alternative to DQN for FIGARO, which can be explored in future framework improvements. Another relevant suggestion from [24] is that integrating graph neural networks could possibly lead to enhanced performance. Similarly, [11] addresses cost optimization in cloud computing by developing a scheduling approach based on deep RL that balances the use of spot and on-demand instances. Integrated into the Argo workflow engine, an open-source container-native workflow engine for orchestrating parallel jobs on Kubernetes [1], this method achieves great cost savings and more intelligent scheduling strategies.

We can also consider the problem of including edge systems in the pool of available computational resources. [17] introduces a task scheduling system for resource utilization based on DQN, called DQNRTS. This algorithm tackles task scheduling in resource-constrained environments by leveraging deep RL for intelligent task offloading decisions; this approach significantly reduces latency and optimizes resource use. The problem of optimizing service migration paths in Mobile Edge Computing (MEC) environments is addressed by [4] presenting SMART, a method based on deep RL with DQN, able to select optimal migration paths with improvements in latency and cost metrics compared to traditional methods. [5] offloads tasks in MEC with a distributed dynamic algorithm that combines deep RL and queuing models. The approach effectively reduces energy consumption and task delays, maintaining robustness even under high workload conditions.

Federated learning and distributed RL approaches have also been applied to resource management with great results. [6] proposes a method that incorporates federated deep RL for computation offloading and resource allocation in multi-edge smart communities. This method achieves superior performance in delay and energy consumption metrics compared to benchmark methods, thanks to the enhancement of local model updates and the optimization of resource allocation brought by the distributed implementation. Likewise, [29] presents an algorithm that combines deep RL with federated learning for task offloading in harsh MEC environments. This approach not only improves convergence speed but also reduces costs and energy consumption, demonstrating the advantages of federated methods in dynamic and resource-constrained settings.

Various frameworks and algorithms address the challenges of application placement and scheduling in heterogeneous environments. For example, [9] introduces Darly, a scheduler based on deep RL for serverless video analytics applications. By monitoring performance counters and optimizing resource allocation, Darly manages QoS and energy consumption well, outperforming traditional Kubernetes-based methods. [3] presents DRPC, a distributed resource provisioning framework using the Twin Delayed Deep Deterministic Policy Gradient algorithm. This increases the accuracy of the autoscaling process and greatly reduces response times in microservice clusters, although network usage is increased due to the implementation of distributed learning.

Authors in [13] explore the optimization of hierarchical federated learning. They propose an inference load-aware orchestration scheme that reduces latency and communication costs by optimizing node placement and aggregator associations. [7] introduces an intent-driven approach for the orchestration of serverless applications based on RL, which significantly reduces QoS violations and operational costs by focusing on minimizing communication overhead and improving overall energy efficiency. Then, [25] introduces RLQ, a system designed to optimize task allocation in distributed workload queues by framing the problem as a Markov Decision Process. RLQ demonstrates significant improvements in execution costs and times compared to traditional methods and state-of-the-art solutions.

Overall, these research papers are relevant examples of the great potential of RL in the optimization process of the RS-CP problem. In fact, they highlight the implementation of various solutions in diverse computing environments across the continuum, from edge devices to cloud servers to FaaS systems. RL algorithms and techniques are constantly evolving, and they become applicable to increasingly complex and dynamic scenarios.

7 Conclusions and Future Work

This paper presents the advancements in the development of FIGARO, a novel Reinforcement Learning-based framework capable of handling the runtime adaptation of a computing continuum environment in terms of allocated resources for AI inference pipelines. We implemented a hierarchical RL system in which the master agent overlooks the system, managing local and global constraints, while a local agent is deployed per each computational layer. This greatly increases the ability of agents to reach policy convergence with respect to the previous implementation since each agent needs to solve a simpler problem. We also boosted the ability of agents to learn an optimal policy thanks to Behavioral Cloning: they can rely on the experience of an expert, which is SPACE4AI-D, a state-of-the-art tool to find the placement of minimum cost with performance guarantees at design time. Finally, we validate the work considering deployments including up to four components and an inference pipeline, proving the effectiveness of the implementation - which reaches at most 1.4% of QoS constraints violations. In the future, we are planning to integrate the agents in a simulation environment to test their ability to adapt to long-tail/heavy-tail distributions that characterize real systems. Finally, to evaluate the complete framework, we plan to implement the coordinator first and then evaluate our system in a real prototype environment.

Acknowledgements

This work has been (partially) supported by the Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 - Next Generation EU (NGEU).

References

1. Argo Workflows — argoproj.github.io. https://argoproj.github.io/workflows/, [Accessed 06-09-2024]
2. Artificial Intelligence Market Size & Trends, Growth Analysis, Forecast [2030] — marketsandmarkets.com. https://www.marketsandmarkets.com/Market-Reports/artificial-intelligence-market-74851580.html, [Accessed 11-09-2024]

3. Bai, H., Xu, M., Ye, K., Buyya, R., Xu, C.: Drpc: Distributed reinforcement learning approach for scalable resource provisioning in container-based clusters. IEEE Transactions on Services Computing (2024)
4. Cao, B., Ye, H., Liu, J., Tang, B., Tao, Z., Deng, S.: Smart: Cost-aware service migration path selection based on deep reinforcement learning. IEEE Transactions on Intelligent Transportation Systems (2024)
5. Chai, Z., Hou, H., Li, Y.: A dynamic queuing model based distributed task offloading algorithm using deep reinforcement learning in mobile edge computing. Applied Intelligence **53**(23), 28832–28847 (2023)
6. Chen, Z., Xiong, B., Chen, X., Min, G., Li, J.: Joint computation offloading and resource allocation in multi-edge smart communities with personalized federated deep reinforcement learning. IEEE Transactions on Mobile Computing (2024)
7. Filinis, N., Tzanettis, I., Spatharakis, D., Fotopoulou, E., Dimolitsas, I., Zafeiropoulos, A., Vassilakis, C., Papavassiliou, S.: Intent-driven orchestration of serverless applications in the computing continuum. Future Generation Computer Systems **154**, 72–86 (2024)
8. Filippini, F., Cavadini, R., Ardagna, D., Lancellotti, R., Russo Russo, G., Cardellini, V., Lo Presti, A.: FIGARO: reinForcement learnInG mAnagement acRoss computing cOntinua. In: 3rd Workshop on Distributed Machine Learning for the Intelligence Computing Continuum at ACM/IEEE UCC 2023 (to appear) (2023)
9. Giagkos, D., Tzenetopoulos, A., Masouros, D., Soudris, D., Xydis, S.: Darly: Deep reinforcement learning for qos-aware scheduling under resource heterogeneity optimizing serverless video analytics. In: 2023 IEEE 16th International Conference on Cloud Computing (CLOUD). pp. 1–3. IEEE (2023)
10. Graesser, L., Keng, W.L.: Foundations of Deep Reinforcement Learning: Theory and Practice in Python. Addison-Wesley Professional, 1st edn. (2019)
11. Jayanetti, A., Halgamuge, S., Buyya, R.: A deep reinforcement learning approach for cost optimized workflow scheduling in cloud computing environments. arXiv preprint arXiv:2408.02926 (2024)
12. Jayanetti, A., Halgamuge, S., Buyya, R.: Reinforcement learning based workflow scheduling in cloud and edge computing environments: A taxonomy, review and future directions. arXiv preprint arXiv:2408.02938 (2024)
13. Lackinger, A., Frangoudis, P.A., Čilić, I., Furutanpey, A., Murturi, I., Žarko, I.P., Dustdar, S.: Inference load-aware orchestration for hierarchical federated learning. arXiv preprint arXiv:2407.16836 (2024)
14. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc. (1984)
15. Li, S., Li, B., Zhao, W.: Joint optimization of caching and computation in multi-server NOMA-MEC system via reinforcement learning. IEEE Access **8**, 112762–112771 (2020)
16. Li, Y., Zhang, X., Zeng, T., Duan, J., Wu, C., Wu, D., Chen, X.: Task placement and resource allocation for edge machine learning: A GNN-based multi-agent reinforcement learning paradigm (2023)
17. Luo, C., Zhang, J., Cheng, X., Hong, Y., Chen, Z., Xing, X.: Computation offloading in resource-constrained edge computing systems based on deep reinforcement learning. IEEE Transactions on Computers (2023)
18. Ray: RLLib: Industry-Grade Reinforcement Learning. https://docs.ray.io/en/latest/rllib/index.html, accessed: 17/9/2024

19. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You Only Look Once: Unified, Real-Time Object Detection (2016), https://arxiv.org/abs/1506.02640
20. Risco, S., Moltó, G., Naranjo, D.M., Blanquer, I.: Serverless workflows for containerised applications in the cloud continuum. J. Grid Comput. **19**(3), 1–18 (2021)
21. Ross, S., Bagnell, D.: Efficient Reductions for Imitation Learning. In: Teh, Y.W., Titterington, M. (eds.) Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. Proceedings of Machine Learning Research, vol. 9, pp. 661–668. PMLR, Chia Laguna Resort, Sardinia, Italy (13–15 May 2010), https://proceedings.mlr.press/v9/ross10a.html
22. Rossi, F., Nardelli, M., Cardellini, V.: Horizontal and vertical scaling of container-based applications using reinforcement learning. In: IEEE CLOUD'19. pp. 329–338 (2019). https://doi.org/10.1109/CLOUD.2019.00061
23. Sedghani, H., Filippini, F., Ardagna, D.: A random greedy based design time tool for AI applications component placement and resource selection in computing continua. In: IEEE EDGE'21. pp. 32–40 (2021)
24. Song, A., Lazarev, N., Gohil, V., Li, Y.: Scarlet: Serverless container autoscaling with reinforcement learning environments
25. Staffolani, A., Darvariu, V.A., Bellavista, P., Musolesi, M.: Rlq: Workload allocation with reinforcement learning in distributed queues. IEEE Transactions on Parallel and Distributed Systems **34**(3), 856–868 (2023)
26. Su, Y., Fan, W., Gao, L., Qiao, L., Liu, Y., Wu, F.: Joint DNN Partition and Resource Allocation Optimization for Energy-Constrained Hierarchical Edge-Cloud Systems. IEEE Trans. Veh. Technol. **72**(3), 3930–3944 (2023)
27. Valmari, A.: The state explosion problem. In: Advanced Course on Petri Nets. pp. 429–528. Springer (1996)
28. Wei, W., Gu, H., Wang, K., Li, J., Zhang, X., Wang, N.: Multi-dimensional resource allocation in distributed data centers using deep reinforcement learning. IEEE Trans. Netw. Service Manag. **20**(2), 1817–1829 (2023)
29. Xiang, H., Zhang, M., Jian, C.: Federated deep reinforcement learning-based online task offloading and resource allocation in harsh mobile edge computing environment. Cluster Computing **27**(3), 3323–3339 (2024)