FIGARO: reinForcement learnInG mAnagement acRoss the computing cOntinuum

Federica Filippini
Riccardo Cavadini
Danilo Ardagna
federica.filippini@polimi.it
riccardo.cavadini@polimi.it
danilo.ardagna@polimi.it
Politecnico di Milano
Milano, Italy

Riccardo Lancellotti
riccardo.lancellotti@unimore.it
Università degli Studi di Modena e Reggio Emilia
Modena, Italy

Gabriele Russo Russo
Valeria Cardellini
Francesco Lo Presti
russo.russo@ing.uniroma2.it
cardellini@ing.uniroma2.it
lopresti@info.uniroma2.it
Università di Roma Tor Vergata
Roma, Italy

Abstract

The widespread adoption of Artificial Intelligence applications to analyze data generated by Internet of Things sensors leads to the development of the edge computing paradigm. Deploying applications at the periphery of the network effectively addresses cost and latency concerns associated with cloud computing. However, it generates a highly distributed environment with heterogeneous devices, opening the challenges of how to select resources and place application components. Starting from a state-of-the-art design-time tool, we present in this paper a novel framework based on Reinforcement Learning, named FIGARO (reinForcement learnInG mAnagement acRoss the computing cOntinuum). It handles the runtime adaptation of a computing continuum environment, dealing with the variability of the incoming load and service times. To reduce the training time, we exploit the design-time knowledge, achieving a significant reduction in the violations of the response time constraint.

CCS Concepts: • **Computing methodologies → Artificial intelligence**; **Reinforcement learning**; • **Computer systems organization → Distributed architectures**.

Keywords: Reinforcement Learning, Computing Continuum, Artificial Intelligence

ACM Reference Format:
Federica Filippini, Riccardo Cavadini, Danilo Ardagna, Riccardo Lancellotti, Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. 2023. FIGARO: reinForcement learnInG mAnagement acRoss the computing cOntinuum. In *2023 IEEE/ACM 16th International Conference on Utility and Cloud Computing (UCC '23), December 4–7, 2023, Taormina (Messina), Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3603166.3632565

1 Introduction

The *cloud computing* paradigm is widely adopted [5] and effectively supports the execution of resource-demanding Artificial Intelligence (AI) applications by making an ideally unlimited computational and storage power accessible according to pay-to-go pricing models. The last few years witnessed an accelerated migration towards mobile computing and Internet of Things [11], motivating the rise of a new *edge computing* framework, where high benefits can be achieved by exploiting resources at the periphery of the network. Edge resources, however, have usually less computing capacity than cloud ones and easily become a bottleneck in the computation. Therefore, a *computing continuum* paradigm is emerging: latency-sensitive tasks can be distributed among edge nodes, while computing intensive tasks are offloaded to the cloud layers [20, 3]. The Resource Selection and application Components Placement (RS-CP) problem in the computing continuum is challenging, since the available resources are highly heterogeneous. Moreover, offloading decisions have a significant impact on the system costs, which need to be balanced with security, latency and privacy constraints. Many literature proposals (e.g., [14, 15]) tackle the RS-CP problem at design time, determining the minimum-cost solution that satisfies Quality of Service (QoS) constraints. Despite the good results they achieve, design-time tools suffer from great limitations: to identify a solution in a reasonable computing time, they need to follow some assumptions on, e.g., the input workload of the AI applications under study, network connectivity, distribution of components service times. All these characteristics are usually subject to fluctuations in real systems and can hardly be controlled or even predicted in practice. Therefore, the initial, design-time solution needs to be adapted at runtime to limit the risk of resource saturation or underutilization.

This paper presents FIGARO (reinForcement learnInG mAnagement acRoss the computing cOntinuum), a runtime framework based on Reinforcement Learning (RL) that automatically learns to solve the RS-CP problem under varying system conditions. RL has recently become popular in addressing the placement problem in the computing continuum, as the considered environment is highly dynamic, usually too complex to be effectively modeled in a closed-form and involves high-dimensional states [22]. To reduce the training time required by the RL agent, we designed an initial offline training period where FIGARO learns from the insights provided by the design-time tool in [14]. The partially-trained agent enters then a validation loop where it interacts with an environment simulator and uses the new experience to refine its policy in an infinitely-evolving scenario.

The paper is organized as follows: Section 2 describes the RS-CP problem. Section 3 presents a sample AI workflow. Section 4 reframes the RS-CP problem to be exploited in the context of Reinforcement Learning. Section 5 describes the FIGARO framework, whose experimental validation is discussed in Section 6. Section 7 briefly overviews the state of the art. Conclusions are drawn in Section 8.

2 RS-CP Problem

The general goal of the RS-CP problem is to minimize the execution costs of the computing continuum resources, while complying with memory constraints, components-to-resources compatibility and QoS requirements. Solving it at design time means choosing the most appropriate resources at each computational layer in the continuum, and determining where to execute the AI application components. At runtime, the design-time solution is considered as a starting point, and the resources and components deployments are re-evaluated in the new environment conditions.

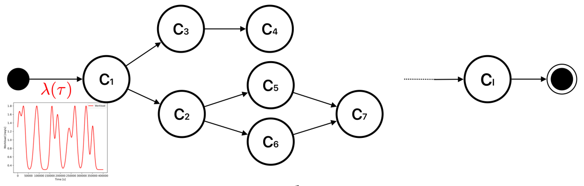


Figure 1. Application DAG.

Following the approach described in [14] for the design-time scenario, we model AI applications as Directed Acyclic Graphs (DAGs) whose nodes represent the different application components (see Figure 1), denoted by the set $\mathcal{I}$. We assume each component $i \in \mathcal{I}$ to be a Deep Neural Network (DNN) running in a software container, which can, in principle, be deployed on heterogeneous resources including edge devices (denoted by the set $\mathcal{J}_{\mathcal{E}}$), cloud Virtual Machines (VMs) (denoted by $\mathcal{J}_C$) and Function as a Service (FaaS) instances ($\mathcal{J}_{\mathcal{F}}$). Each DAG has a single entry point, which receives in every time-interval τ an exogenous input workload $\lambda(\tau)$ (expressed in terms of requests per second), and a single exit point. Each edge is characterized by the transition probability between application components.

DNNs might be partitioned, so that different groups of layers are executed in a distributed manner. A given DNN may be partitioned differently, according to resources capacity and network settings. Thus, each component is characterised by a set C^i of candidate deployments, such that $c^i_s \in C^i$ is a set of neural network partitions, i.e., $c^i_s = \{\pi^i_h\}_{h \in \mathcal{H}^i_s}$.

QoS requirements might be imposed on the response times **R** of single components (local constraints) or on paths including multiple components (global constraints). For components deployed on edge devices or cloud VMs, these response times were computed at design time exploiting M/M/1 queues, i.e., relying on the assumption that inter-arrival and service times follow an exponential distribution. As we will discuss later, this assumption is relaxed at runtime, and we will rely on a simulator that allows to consider heterogenous service times distributions. Furthermore, executing consecutive components or partitions on different resources requires to transfer data 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 characterised by hourly costs that account for resources and energy consumption. FaaS costs are expressed in GB per second and are related to the memory configuration of the corresponding instances. These costs are denoted by C_E, C_C and C_F, respectively.

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 [14]):

$$\min C_E + C_C + C_F. \tag{1}$$

Solving this problem at runtime means switching on/off the appropriate edge resources, scaling in/out the cloud VMs instances, identifying a deployment for each component, allocating the partitions on the chosen devices and checking the compatibility with memory constraints and QoS requirements. The choice of the DNN deployment for each component is managed through the binary vector **Z**, whose element z^i_s is 1 if c^i_s is selected for component $i \in \mathcal{I}$. Similarly, the component-to-resource assignments are described through the matrix $\hat{\mathrm{Y}}$. Each element $\hat{y}^i_{hj}$ is equal to the number of instances of resource $j \in \mathcal{J}$ assigned to partition π^i_h.

To effectively tackle the RS-CP problem at runtime, dynamically adapting to varying environment conditions, we reformulated this MINLP as a Markov Decision Process and address it by developing a Reinforcement Learning-based approach, as described in Sections 4 and 5.

3 Sample Use-Case Application

Consider the sample AI inference application illustrated in Figure 2. Initially proposed in [12], it consists of a simple workflow including two components designed to monitor

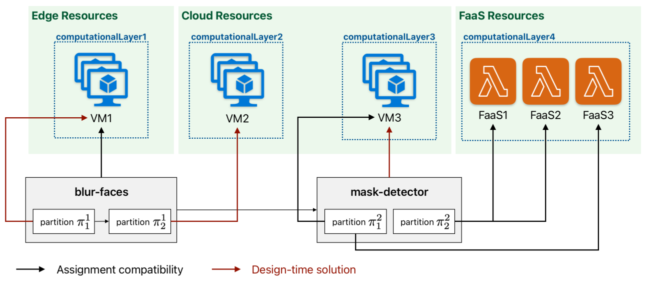


Figure 2. Sample AI workflow with partitionable components and multiple candidate computational layers.

videos produced by surveillance cameras in various city areas and determining which ones are characterized by the highest percentage of people not wearing face-masks. In particular, *blur-faces* splits the input video in a sequence of frames, processes each image to detect faces and blurs the corresponding areas to guarantee privacy preservation. On the other hand, *mask-detector* works on the anonymized images to identify and classify each face according to the presence of face masks. Both components include an object-detection task performed through a DNN that can be differently partitioned. In the example of Figure 2, for instance, we assume that both *blur-faces* and *mask-detector* can be considered either as unique, non-partitioned components or as sequences of two partitions, denoted as π_1^1 and π_2^1 for *blur-faces*, and as π_1^2 and π_2^2 for *mask-detector*, respectively. Furthermore, we identify the set of candidate resources that can be considered to execute each component (or component partition). In particular, the whole *blur-faces* component, as well as its first partition, can be executed on a single VM type, denoted by $VM1$ and hosted on a private Edge cluster. Similarly, π_2^1 can be deployed on a single Cloud VM type, denoted by $VM2$. More alternatives are instead available for the second component, since: (i) the whole *mask-detector* can be executed only on a VM type $VM3$ in the Cloud, but (ii) partition π_1^2 can be deployed either on $VM4$ or on an AWS Lambda function configuration denoted as $FaaS3$, and (iii) π_2^2 can be executed on two candidate function configurations ($FaaS1$ or $FaaS2$, respectively), characterized by, e.g., different memory settings.

Finally, we represent in Figure 2 the design-time solution through red arrows connecting the selected component/partitions and the resource they are deployed onto.

4 Reinforcement Learning Problem

RL algorithms are characterized by the presence of an *agent* that, in each time-window τ, learns how to map the current state $s(\tau)$ into an action $a(\tau)$ that maximizes a numeric reward signal $r(\tau)$. In our problem, since the main goal is to minimize the system's operational cost, we substitute this reward with a cost $c(\tau)$ defined so that $c(\tau) = -r(\tau)$. The learning process is guided by the interaction with the environment, which reacts to each agent action by transitioning from the state $s(\tau)$ to $s' = s(\tau')$, where τ' denotes the next time window. The cost associated to each state-action pair depends on this transition, i.e., $c = c(s, a, s')$. The strategy followed by the agent, i.e., the mapping that selects the next action as a function of the current state, is called *policy*. In the following, we characterize the state space $\mathcal{S}$, the action space $\mathcal{A}$, and the cost function $c(s, a, s')$ for our problem.

State Space. In every time-window τ, each state $s(\tau) \in \mathcal{S}$ can be represented as $s(\tau) = \langle \mathbf{Z}, \hat{\mathbf{Y}}, \lambda, \mathbf{R} \rangle$.

Note that, since the input workload $\lambda(\tau)$ and the response times $\mathbf{R}$ are continuous variables, traditional tabular RL methods that use matrices to associate a specific state or state-action pair to the expected reward cannot be applied in this context [17]. Even for small systems involving two application components, a discretization of the λ and $\mathbf{R}$ variables considering 10 values would easily result in a state space dimension close to 10^9 (i.e., a memory requirement of nearly $4GB$), which is unfeasible in practical scenarios. We will therefore consider the Deep Q-Learning algorithm, as discussed in Section 5.

Action Space. The available actions selected by the agent according to the developed policy may involve single components or resources. Specifically, the agent can: select a different DNN deployment for a given component, scale in/out the number of VM instances selected for a given resource or migrate a component partition on a different resource with respect to the one where it is currently being executed. In addition to these reconfiguration actions, the action set $\mathcal{A}(s)$ comprises a "do-nothing" action η for every state s. When the agent picks the action η, the system configuration is unchanged. Note that, to keep the cardinality of $\mathcal{A}$ under control, only atomic moves are allowed, meaning that, for instance, the agent cannot decide to concurrently scale in a VM resource and migrate a component partition.

Cost Function. Our cost $c(s, a, s')$ in Equation 2 includes four main components: $\kappa^{GC}(s')$ and $\kappa^{LC}(s')$ are performance penalties related to the violation of Global or Local QoS Constraints, respectively. $\kappa^{exec}(s, a, s')$ is the execution cost of the component partitions on the selected resources, and it corresponds to the total cost defined in Equation (1). Finally, $\kappa^{rcf}(a)$ is a penalty cost related to system reconfigurations, which is introduced to account for the delays incurred when, e.g., increasing the number of selected VM instances or stopping and restarting components to perform migrations. We defined κ^{GC}, κ^{LC} and κ^{rcf} as binary costs, meaning that κ^{GC} and κ^{LC} are equal to 1 if any global or local constraint is violated, respectively, while κ^{rcf} is 1 if the agent chooses any action different from η. The relative importance of these four terms is determined through suitable weights w^{GC}, w^{LC}, w^{exec} and $w^{rcf} \in [0, 1]$ and such that $\sum_l w^l = 1$.

The cost function $c(s, a, s')$ is defined as in other proposals [4] by the Simple Additive Weighting approach:

$$c(s,a,s') = w^{GC}\kappa^{GC}(s') + w^{LC}\kappa^{LC}(s') + w^{exec}\frac{\kappa^{exec}(s,a,s') - C^{\min}}{C^{\max} - C^{\min}} + w^{rcf}\kappa^{rcf}(a). \quad (2)$$

$C^{\min}, C^{\max} \in \mathbb{R}^+$ are cost-normalization constants, which can be computed *a priori* by assuming to select the cheapest or most expensive assignment for each component partition.

Note that adopting binary penalties for the QoS constraints violations, neglecting the number or entity of the violations, may be a strong limitation in practice. In future work we plan to address this issue so to better penalize the delays.

5 FIGARO

FIGARO (reinForcement learnInG mAnagement acRoss the computing cOntinuum) is an integrated framework that supports the runtime management of AI applications. One of the main sources of variability in a real environment is the input workload. This usually fluctuates over time due to, e.g., variations in the incoming data volumes, increasing the risk of resources saturation or underutilization if a static assignment is considered. Periodically running the design-time framework to determine a new optimal solution is not feasible, since this usually requires a too large computing time to be exploited at runtime (where reconfigurations need to happen in seconds). RL algorithms, which continuously learn how to adapt their choices according to the environment responses, are very promising in this field. However, they usually require a significant training time to learn an effective behavior. To mitigate this issue, which would lead, in the initial exploratory phase, to frequent QoS violations or large overspending, we designed the FIGARO framework to exploit the design-time knowledge available from the open-source SPACE4AI-D tool available in the literature [14]. Our RL agent learns how to mimic the design-time policy, which can be considered as a good starting point for runtime reconfigurations, in an initial, offline training phase. The learned policy is then injected in a simulation loop where the interactions with the Environment Simulator are used for validation and may determine further updates according to the new experienced scenarios.

The goal of the offline training loop is to let the Agent learn an initial policy that can achieve results comparable to SPACE4AI-D. During this training stage, our Agent does not interact with the Environment Simulator: the response times of all components are estimated via M/M/1 models.

The FIGARO architecture is illustrated in Figure 3 and comprises the following components:

- **Workload Trace Generator**: it simulates a workload injector, generating a trace for $\lambda(\tau)$.
- **Coordinators**: the goal of both the Offline Training Coordinator (OTC) and the Simulation Loop Coordinator (SLC) is to manage the communications among other components. In each time-window τ, they query the Workload Trace Generator to retrieve $\lambda(\tau)$. Then:
  - The OTC queries SPACE4AI-D, which returns a good-quality configuration used as baseline to evaluate the improvement of our RL Agent. Based on the difference between the current Agent policy and the behavior of SPACE4AI-D, it exploits HyperOpt [2] to determine the next set of hyperparameters, provided to the Agent to start the new training loop.
  - The SLC queries the Environment Simulator to retrieve the components response times $\mathbf{R}(\tau)$ under the current configuration $\langle\hat{\mathbf{Y}}(\tau), \mathbf{Z}(\tau)\rangle$ and workload $\lambda(\tau)$, computes the cost accordingly and provides this information to the Agent, receiving in turn the next configuration to be considered.
- **Environment Simulator**: given the current configuration $\langle\hat{\mathbf{Y}}(\tau), \mathbf{Z}(\tau)\rangle$ and input workload $\lambda(\tau)$, it returns the response times $\mathbf{R}(\tau)$ of all the application components.
- **Agent**: given the current configuration $\langle\hat{\mathbf{Y}}(\tau), \mathbf{Z}(\tau)\rangle$, input workload $\lambda(\tau)$, response times $\mathbf{R}(\tau)$ and observed cost, it determines the next action to be applied in the system, thus generating a new solution. As discussed in the following, the agent can implement a static policy or dynamically learn from the observed interactions.

Offline Training. The FIGARO Agent is based on the Deep Q-Learning (DQL) algorithm. Q-Learning is an off-policy RL algorithm aiming to learn the optimal action-value function $Q^*(s,a)$ [17], whose current approximation is stored in a table and updated every time a new pair (s,a) is visited. The agent chooses the next action at each step according to the so-called ε-greedy policy, i.e., by selecting with probability $\varepsilon \in [0,1]$ a random action, and with probability $1-\varepsilon$ the action a' that maximizes the current value $Q(s,a')$. The DQL algorithm extends Q-Learning to more complex scenarios, where a tabular representation of the action-value function is not feasible (e.g., because the state or action space is infinite-dimensional) [8], approximating $Q(s,a)$ through a Deep Neural Network called Q-Network. We adopted DQL in our problem since, as discussed in Section 4, our state space includes continuous variables.

The goal of the offline training loop is to let the Agent learn an initial policy that can achieve results comparable to SPACE4AI-D. During this training stage, our Agent does not interact with the Environment Simulator: the response times of all components are estimated via M/M/1 models.

Due to the wide range of hyperparameters that characterize DQL algorithms, the offline training proves to be a complex task, requiring a large computational time to be executed effectively (e.g., in our experimental setting, a training loop with nearly 1000 iterations could last up to three days). HyperOpt helps to speed-up the parameters space exploration by leveraging Bayesian optimization to find the most

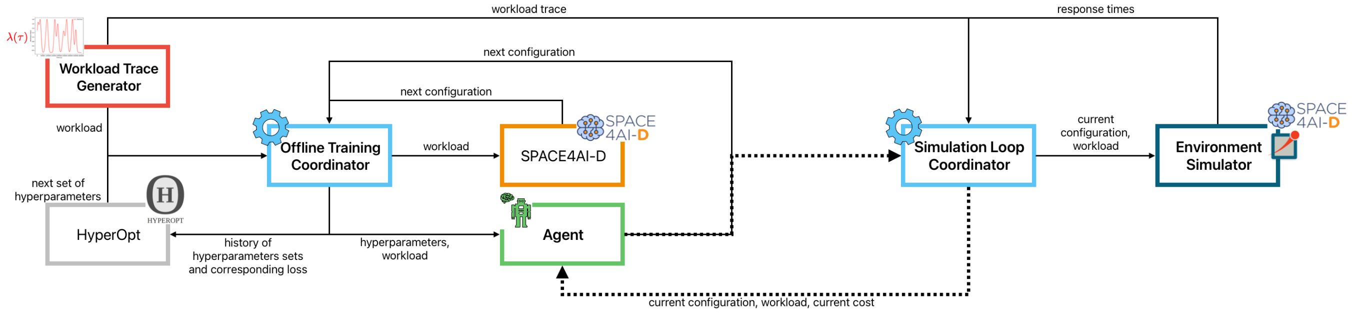

workload trace
response times
$\lambda(\tau)$
Workload Trace Generator
next configuration
next configuration
workload
Offline Training Coordinator
workload
SPACE4AI-D
SPACE 4AI-D
current configuration, workload
Simulation Loop Coordinator
Environment Simulator
SPACE 4AI-D
next set of hyperparameters
HYPEROPT
HyperOpt
history of hyperparameters sets and corresponding loss
hyperparameters, workload
Agent
current configuration, workload, current cost

Figure 3. FIGARO architecture.

promising set of hyperparameters, comparing the current policy with the behavior of SPACE4AI-D.

Simulation Loop. The learned policy is evaluated and, possibly, further updated by connecting the Agent to the SLC (dotted lines in Figure 3), creating a new loop featuring the interactions with the Environment Simulator. This can be implemented exploiting: (i) SPACE4AI-D, which computes the response times through analytical M/M/1 models (see Section 2), or (ii) a simulator based on OMNeT++, which supports heterogeneous service times distributions. Interactions with a real environment is part of our future work.

In the initial policy-evaluation phase, the Agent plays a *static* policy, i.e., the new collected experience is not used to trigger further updates. However, continuous learning is one of the crucial benefits of RL; FIGARO can leverage the state-action pairs observed while interacting with the Environment Simulator to improve a *dynamic* ε-greedy policy.

6 Experimental Results

We now present the experimental evaluation of FIGARO. As this is the first implementation of our framework, we consider application pipelines including up to two components, as the one presented in Section 3, and enable only scaling actions. We set a response time constraint for the whole application, and, since our main goal is to avoid violations, we chose the weights in Equation (2) as: $w^{GC} = 0.9$, $w^{exec} = 0.08$, $w^{rcf} = 0.02$, as discussed in, e.g., [4, 13] (w^{LC} is 0 since we do not consider local constraints). Table 1 lists the parameters used to train the Agents (both offline and online). They can be divided into environment parameters (weights for constraints and costs, description of workload, and of episodes), network parameters (to describe the network used for the DQN algorithm), and agent + training parameters (description of how the agent should behave). We chose the optimal parameters for our experimental setting according to Hyperopt [2], which was originally fed with the ranges shown in the second column of Table 1.

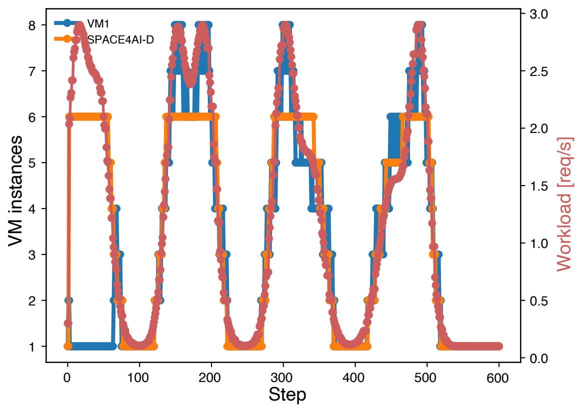
Each component is deployed on a fixed VM. At this stage of proof of concept, only atomic scaling actions are available: increase/decrease by 1 the number of instances of a VM, or keep the configuration unchanged.

Table 1. Parameters used to train the Agents.

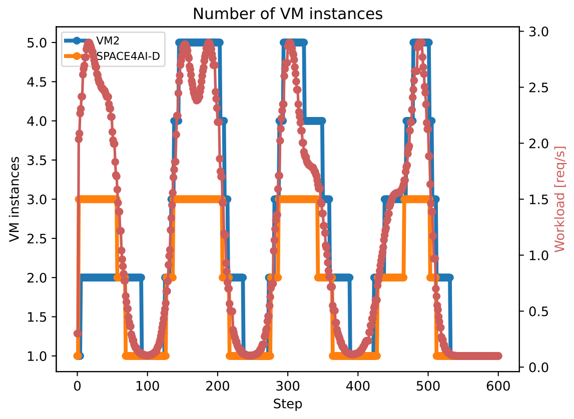
Name	Range	Optimal Value
EnvironmentParameters/ObjectiveFunctionWeights		
GlobalConstraintWeight	–	0.899999
LocalConstraintWeight	–	0.000001
ExecutionCosts	–	0.08
ReconfigurationCosts	–	0.02
EnvironmentParameters/WorkloadVariation		
Type	–	exponential
AverageInterval	30, 60, 120, 240, 360, 720	60
MaxNVariations	150, 300, 600, 900, 1200, 2400	1200
MinNBimodal	2, 4, 5, 6, 7, 8, 10	4
BimodalInterval	–	15000
EnvironmentParameters		
EpisodeLength	–	60000
ReconfigurationInterval	50, 100, 200, 500, 800	100
BatchSize	1, 2, 4, 10, 20, 50	4
NetworkParameters		
ActivationFunction	*linear*, *relu*, *sigmoid*, *tanh*	tanh
NumberOfLayers	1, 2, 3, 5	2
NumberOfNeurons	30, 50, 75, 100, 125, 200	75
QLayerActivationFunction	–	linear
AgentParameters		
LearningRate	0.0005, 0.001, ..., 0.004	0.001
Optimiser	–	adam
Epsilon	0.05, 0.1, ...0.35, 0.4	0.15
TargetUpdateTau	0.05, 0.1, ...0.35, 0.4	0.2
TargetUpdatePeriod	10, 40, 70, ..., 210	150
TDerrorsLossFunction	–	element-wise squared loss
Gamma	–	0.99
EpsilonDecay	0.7, 0.75, ..., 0.95, 1	1.0
LearningRateDecay	0.7, 0.75, ..., 0.95, 1	0.95
TauDecay	0.7, 0.75, ..., 0.95, 1	0.9
TrainingParameters		
BatchSize	32, 64, 128, 256, 512, 1024	64
ReplayBufferCapacity	1, 2, ..., 10 $*$ $MaxNVariations$	2400
NumberCollectedEpisodes	1, 2, 3, 5, 10	3

We generated a realistic profile for $\lambda(\tau)$. Request rates usually follow some patterns, such as bell-shaped curves with a bi-modal distribution. The Workload Trace Generator mimics this trend: we generated a synthetic reference workload from a real web system, scaled between λ_{min} and λ_{max} and modified by adding noise a random shift in time.

We first discuss the results obtained during the offline training of the RL Agent in the case of a two-component application (a single-component application shows a similar behavior). Figure 4 reports the number of instances selected by the policy for the two VMs, over one episode including 600 steps, compared with the results of SPACE4AI-D. We can see that the Agent effectively mimics SPACE4AI-D, following the workload evolution by adapting the number of allocated resources.



(a) Selected VM1 instances



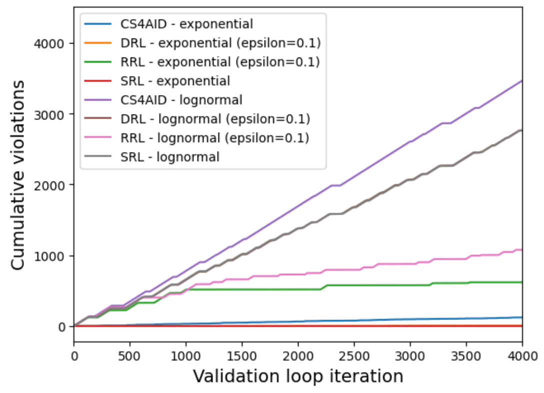
(b) Selected VM2 instances

Figure 4. Number of selected VM instances in a 2-components system. Comparison between the choices made by SPACE4AI-D (orange), those by the RL Agent (blue), and the workload (red).

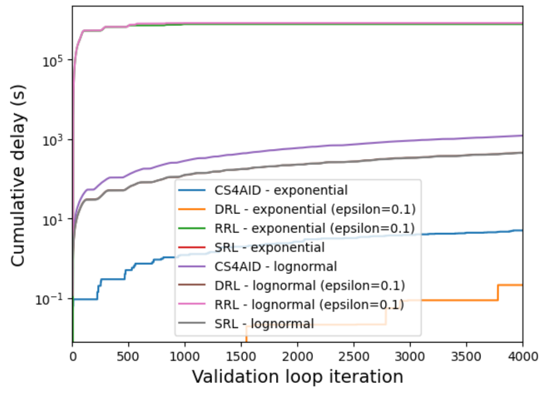
After the offline training terminates, we plugged the final policy in the validation loop. We compared the behavior of various Agents, interacting with the Environment Simulator:

- A Static RL Agent (SRL) that applies the policy learned offline without further improvements;
- A Dynamic RL Agent (DRL) that continuously updates an ε-greedy policy starting from the one learned offline;
- Constrained SPACE4AI-D (CS4AID), which enables the comparison with the design-time tool: when determining the optimal configuration at design-time, SPACE4AI-D has no constraints on the number of decisions it can perform, while the RL Agent can only take atomic actions that involve a single component or resource (see Section 4). CS4AID runs SPACE4AI-D to get the new optimal configuration, but is forced to apply only atomic actions to implement it. Comparing FIGARO with CS4AID highlights whether it can perform better when considering each specific action.
- A Randomly-initialized RL Agent (RRL) implemented to test the benefits of the offline training: it corresponds to DRL but starts from a random policy.

We configured the Environment Simulator to generate response times under three different assumptions on the service-time distributions, to test the Agents reactions in heterogeneous scenarios: (i) exponential with the same mean μ considered during the offline training, (ii) exponential with mean increased by 20%, and (iii) log-normal with mean μ and standard deviation 2μ. To analyze the Agent's performance, we evaluated the number of global constraints violations and the *cumulative delay*, i.e., the cumulative difference across the whole simulation between the response times and the corresponding thresholds.



(a) Cumulative number of violations.

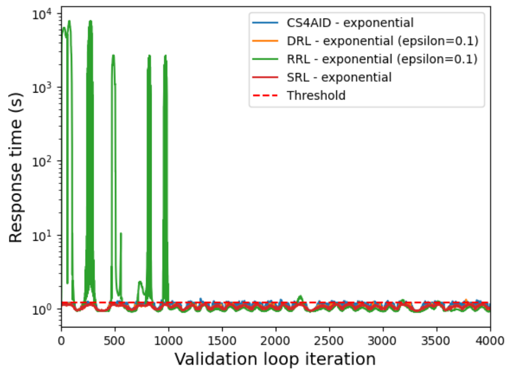


(b) Cumulative delay.

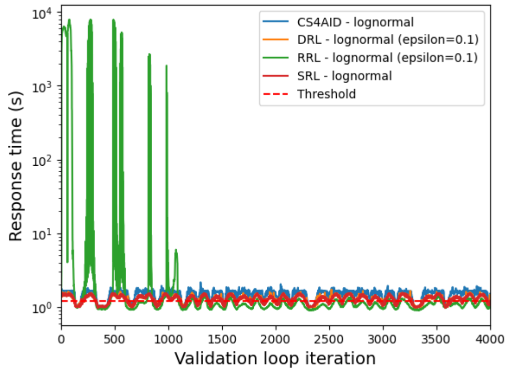
Figure 5. Cumulative number of violations and delay for all Agents, with both exponential and log-normal distributions.

Figure 5 shows the behavior of all Agents for the exponential distribution with mean μ and the log-normal distribution. As expected, the cumulative number of violations in Figure 5a is significantly lower in the exponential case for all the Agents trained offline, which had experienced the same distributions. Moreover, the pre-trained Agents always violate less than CS4AID, proving the efficacy of our method.

While performing much worse than the others with the exponential distribution, RRL shows fewer violations in the log-normal setting. This is due to the fact that it is not biased in the beginning, as it starts from a random policy. Therefore, it is more free to explore and test solutions, learning a valid policy in around 1000 loop iterations and then showing similar results for both distributions. However, as we can see in Figure 5b, this exploration comes at the cost of very large delays. This plot further demonstrates how effective the offline training is, since also CS4AID is responsible for a greater cumulative delay under both distributions. Figure 6 shows the response times of the online system when each policy is applied. The most relevant information represented in these plots is that the Dynamic Agent with a random initialization shows high response times for the first 1000 iterations, after which it learns how to behave properly.



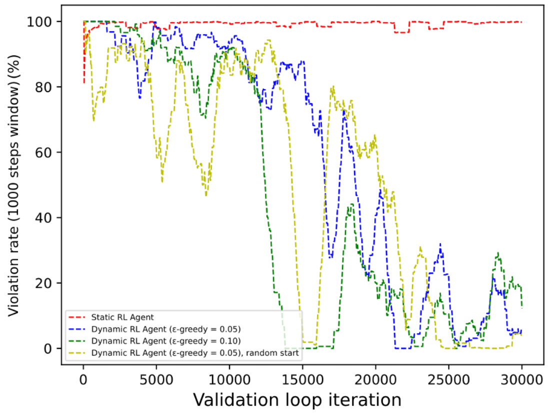
(a) DRL and CS4AID are comparable, while RRL shows very large response times for the first 1000 iterations.



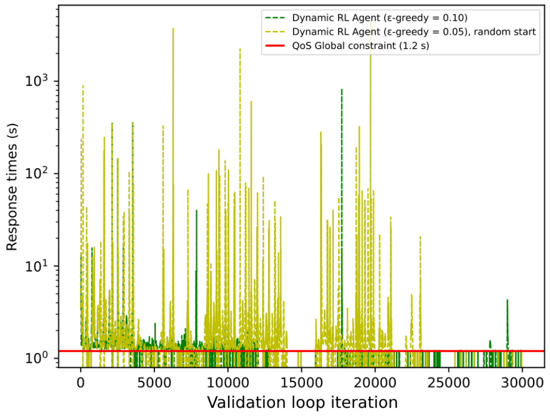
(b) RRL is significantly more inefficient than the others for the first 1000 iterations; for all agents, response times are higher than in the exponential case.

Figure 6. Application response time with components demand following an exponential and a log-normal distribution.

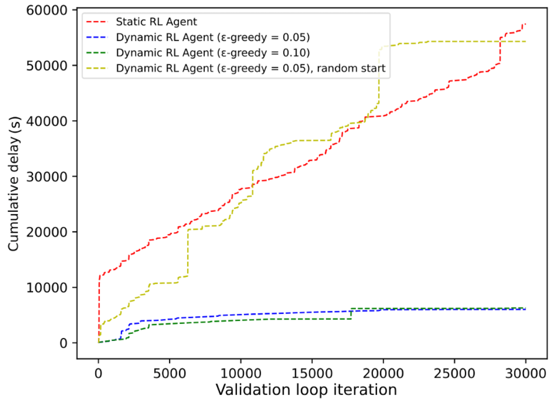
By moving from an exponential to a log-normal distributions we proved that offline training is useful to reduce violations. SRL and DRL show comparable results, which would apparently invalidate the benefits of continuous learning. However, Figure 7 shows a comparison of the behaviors of the Agents when the components demand is increased by 20%. The static agent is the worst performing of all, as its violation rate never changes and is close to 100%. The violation rates of the Dynamic Agents actually decrease throughout the validation iterations, and their behaviors look similar to each other. However, analyzing the application response time it is clear that the Dynamic Agent initialized with the offline policy is performing better than the randomly initialized one. Moreover, the entity of violations of the randomly initialized Agent is in fact much higher throughout the first 20000 iterations.



(a) Violation rate.



(b) Application response time.



(c) Cumulative delay.

Figure 7. Comparison of the behaviors of the Agents when the average service times are increased by 20%.

7 Related Work

Lately, a lot of research has been devoted to the RS-CP problem. Many literature proposals tackle the problem at design time; for example, [14] and [15] determine the minimum-cost solution that complies with QoS restrictions for a fixed value of expected workload.

A work similar to ours is [13], where RL manages the horizontal and vertical elasticity of container-based applications. Being RS-CP a continuous-control problem, it can be tackled efficiently with Deep Reinforcement Learning (DeepRL). DeepRL has been applied to various computing-continuum systems, e.g., UAVs [21], real-time tasks in a Mobile Edge Computing setting [1] and Vehicular Edge Computing [10].

DeepRL for resource allocation has been also applied to Virtual Reality (VR) and eXtended Reality (XR): in [16] and [18], computationally demanding tasks are offloaded at runtime by relying on DeepRL techniques. Moreover, a relevant topic when dealing with Cloud and Fog computing is energy consumption: this is tackled by works such as [9].

Overall, multi-resource allocation problems are not trivial to solve [19]. Recently, authors have addressed them in multi-agent settings where each RL agent can focus just on a specific subtask, such as [7]. Multiple agents can be cooperative or independent. Some solutions have demonstrated the efficacy of a hierarchical collaborative organization [6].

8 Conclusions and Future Work

This paper proposes FIGARO, a novel Reinforcement Learning-based framework capable of handling the runtime adaptation of a computing continuum environment in terms of allocated resources for specific application components. FIGARO exploits the design-time knowledge to speed up the learning process of the Agent, enabling the deployment of a policy that is already effective with respect to the environment. We tested our framework in a simulated environment and validated it by changing the service-time distributions. By measuring the response times of the application components, we demonstrated that our framework outperforms a static agent adapted from the design time, particularly when considering initial offline training and then continuously updating the policy during the simulation.

This work refers to components of an AI application, but the same approach could be extended to cope with the deployment of microservices as well, in particular to those that can be partitioned and for which we might have multiple versions. The current implementation is only capable of performing scaling actions on the number of resource instances, but we plan to integrate the migration of components between different resources. Moreover, we also plan to evaluate our framework in a real environment (instead of a simulated one) to ultimately validate it.

Acknowledgments

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] Zeinab Akhavan et al. 2022. Deep reinforcement learning for online latency aware workload offloading in mobile edge computing. In *IEEE GLOBECOM'22*, 2218–2223.

[2] James Bergstra et al. 2013. Making a science of model search: hyperparameter optimization in hundreds of dimensions for vision architectures. In *ICML'13*. Vol. 28, 115–123.

[3] Zequn Cao and Xiaoheng Deng. 2023. Dependent task offloading in edge computing using GNN and deep reinforcement learning. (2023). arXiv: 2303.17100 [cs.DC].

[4] Valeria Cardellini et al. 2018. Decentralized self-adaptation for elastic data stream processing. *Future Gener. Comput. Syst.*, 87, 171–185.

[5] FORTUNE Business Insights. 2023. Cloud computing market size, growth & COVID-19 impact analysis, 2023–2030. https://www.fortunebusinessinsights.com/cloud-computing-market-102697. (2023).

[6] Shilu Li et al. 2020. Joint optimization of caching and computation in multi-server NOMA-MEC system via reinforcement learning. *IEEE Access*, 8, 112762–112771.

[7] Yihong Li et al. 2023. Task placement and resource allocation for edge machine learning: A GNN-based multi-agent reinforcement learning paradigm. (2023). arXiv: 2302.00571 [cs.MA].

[8] Volodymyr Mnih et al. 2013. Playing Atari with deep reinforcement learning. (2013). arXiv: 1312.5602 [cs.LG].

[9] Malathy Navaneetha Krishnan and Revathi Thiyagarajan. 2022. Multi-objective task scheduling in fog computing using improved gaining sharing knowledge based algorithm. *Concurr. Comput.*, 34, 24, e7227.

[10] Xin Peng et al. 2023. Deep reinforcement learning for shared offloading strategy in vehicle edge computing. *IEEE Syst. J.*, 17, 2.

[11] Edgar Ramos et al. 2019. Distributing intelligence to the edge and beyond [research frontier]. *Comp. Intell. Mag.*, 14, 4, 65–92.

[12] Sebastián Risco et al. 2021. Serverless workflows for containerised applications in the cloud continuum. *J. Grid Comput.*, 19, 3, 1–18.

[13] Fabiana Rossi et al. 2019. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *IEEE CLOUD'19*, 329–338.

[14] Hamta Sedghani et al. 2021. A random greedy based design time tool for AI applications component placement and resource selection in computing continua. In *IEEE EDGE'21*, 32–40.

[15] Yi Su et al. 2023. Joint DNN Partition and Resource Allocation Optimization for Energy-Constrained Hierarchical Edge-Cloud Systems. *IEEE Trans. Veh. Technol.*, 72, 3, 3930–3944.

[16] Yaohua Sun et al. 2022. Enabling mobile virtual reality with open 5G, fog computing and reinforcement learning. *IEEE Network*, 36, 6, 142–149.

[17] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT Press.

[18] Bao Trinh and Gabriel-Miro Muntean. 2022. A deep reinforcement learning-based offloading scheme for multi-access edge computing-supported extended reality systems. *IEEE Trans. Veh. Technol.*, 72, 1, 1254–1264.

[19] Wenting Wei et al. 2023. Multi-dimensional resource allocation in distributed data centers using deep reinforcement learning. *IEEE Trans. Netw. Service Manag.*, 20, 2, 1817–1829.

[20] Somayeh Yeganeh et al. 2023. A novel Q-learning-based hybrid algorithm for the optimal offloading and scheduling in mobile edge computing environments. *J. Netw. Comput. Appl.*, 214, 103617.

[21] Seonghoon Yoo et al. 2023. Hybrid UAV-enabled secure offloading via deep reinforcement learning. *IEEE Wirel. Commun. Lett.*, 12, 6, 972–976.

[22] Zhi Zhou et al. 2021. Deep reinforcement learning for intelligent cloud resource management. In *IEEE INFOCOM’21 Workshops*, 1–6.