Application Component Placement and Resource Optimization in Computing Continua

Hamta Sedghani, Mauro Passacantando, Danilo Ardagna

Abstract—The proliferation of Internet of Things (IoT), Artificial Intelligence (AI), and real-time data processing applications has driven the demand for distributed computing architectures that span cloud, fog, and edge layers in a computing continuum. These architectures must address critical challenges in component placement and resource optimization to ensure low latency, cost efficiency, and compliance with Quality of Service (QoS) constraints. This paper introduces a novel optimization framework for addressing the joint problem of component placement and resource optimization in computing continua. The framework employs a Mixed Integer Nonlinear Programming model, where application components are modeled as a Directed Acyclic Graph, and their performance is predicted using analytical models. A Karush-Kuhn-Tucker (KKT)-based method is employed to compute the optimal number of virtual machine instances for a given component placement. This optimization is embedded within a reinforcement learning (RL) loop that iteratively refines placement decisions in response to workload fluctuations. This hybrid approach ensures cost-effectiveness while adhering to QoS constraints. Extensive experimental evaluations demonstrate the superiority of our framework. It outperforms leading approaches, including BARON solver, SPACE4AI-D, PPO_DLX, and a minimum k-cut baseline, achieving average cost reductions of 19%, 60%, 11%, and 6%, respectively, under dynamic workload conditions. These results highlight the efficiency, scalability, and adaptability of our approach, making it a robust solution for the demands of modern distributed systems.

Index Terms—**Edge, Cloud computing, Component placement, Resource optimization, Reinforcement Learning.**

1 Introduction

The rapid expansion of the Internet of Things (IoT), Artificial Intelligence (AI), and real-time data processing applications has necessitated the development of distributed computing architectures. These architectures span a computing continuum, which integrates centralized cloud resources with decentralized fog and edge computing infrastructures to meet the demands of modern applications that require low-latency, high bandwidth, and energy-efficient processing across multiple layers of the network [1]. As the number of IoT devices grows exponentially, expected to exceed 128 billion by 2027 [2], and the variety of applications that depend on real-time processing proliferates, efficient component placement and resource optimization have become critical challenges that must be addressed to fully exploit the potential of the computing continuum. Specifically, determining how to place software components across the cloud, fog, and edge layers and how to allocate computational resources (e.g., CPU, memory, bandwidth) dynamically and efficiently are central to minimizing the system cost, particularly under Quality of Service (QoS) response time constraints [3], [4]. Poor placement of application components can lead to suboptimal performance, increased latency, and inefficient resource utilization, while inadequate resource allocation can cause system overloads or resource underutilization. Traditionally, resource allocation and component placement problems have been solved using heuristic approaches [5] or by relying on static optimization techniques [6], often as separate problems. While these methods have produced some promising results, they often struggle to adapt to the dynamic nature of the computing continuum, when any change in the system status is observed (e.g., a demand increase or unexpected workload fluctuations) in real time [7].

To overcome these limitations, AI-driven orchestration is emerging as a promising paradigm for managing the complexity and dynamism of the computing continuum. By leveraging predictive analytics, Reinforcement Learning (RL), and online optimization, AI-based orchestrators can make informed decisions in response to real-time changes in workload and network conditions. These systems continuously learn from operational data to anticipate demand fluctuations and optimize deployments with minimal human intervention, thereby improving service continuity, resource efficiency, and compliance with QoS requirements [8].

In this paper, we formulate a joint component placement and resource allocation problem as a Mixed Integer Nonlinear Programming (MINLP) model, where applications are represented as Directed Acyclic Graphs (DAGs) and their performance is evaluated through queuing-theory models. However, solving this MINLP formulation becomes computationally intractable for realistic system scales due to the exponential growth of the placement space and the non-convexity introduced by response time constraints. To address this challenge, we propose a novel optimization framework that partitions the problem into resource allocation and component placement subproblems. Specifically, initially, assuming a fixed placement for all components, we analytically derive the optimal number of virtual machine (VM) instances for each layer of the continuum to meet response time constraints using the Karush Kuhn

- *H. Sedghani and D. Ardagna are with Politecnico di Milano. Milan, Italy. Email: {name.lastname}@polimi.it (Corresponding author: Hamta Sedghani, Email: hamta.sedghani@polimi.it)*
- *M. Passacantando is with University of Milano-Bicocca, Milan, Italy, Email: mauro.passacantando@unimib.it.*

Tucker (KKT) conditions, capitalizing on the structure of the queueing-based model. Subsequently (and by relying on the KKT solution), we solve the component placement problem employing RL, which efficiently explores the extensive discrete action space and identifies effective placement strategies based on environmental feedback. On the one hand, from a theoretical standpoint, this decomposition enables us to utilize convex optimization tools where applicable (resource sizing), while applying learning-based methods where exact solutions are computationally prohibitive (placement). On the other hand, from a practical perspective, the decoupled approach facilitates scalable and modular implementation, which is crucial in large-scale or dynamic computing continua.

In summary, our main contributions are the following:

1) We formulate a Mixed Integer Nonlinear Programming (MINLP) model to address the joint components placement and resource optimization problem.
2) We propose an optimization framework that integrates analytical and learning-based methodologies. By employing the KKT-based approach, we optimize resource allocation under fixed component placement. Subsequently, we leverage RL to adaptively determine optimal placements in response to dynamic system conditions. This hybrid approach ensures both cost-effectiveness and adherence to performance constraints within dynamic computing environments.
3) We evaluated the performance of our resource optimization approach using the BARON [9] state-of-the-art global solver as a benchmark. The experimental results demonstrate that our approach is capable of finding the optimal solution at least 10 times faster than BARON, with a worst-case cost deviation of no more than 2% with respect to BARON. This highlights the efficiency and near-optimality of our method in terms of both speed and cost-effectiveness.
4) We evaluated the performance of our framework against several prominent approaches, including BARON, the SPACE4AI-D framework [10], a recently proposed approach based on RL, and the Dancing Links technique presented in [11] (referred to as PPO_DLX). Furthermore, we benchmarked our framework against a baseline approach based on the minimum k-cut algorithm. Experimental results demonstrate that our framework consistently outperformed these state-of-the-art methods in various dynamic workload scenarios. Under fluctuating system conditions, our framework achieved significant average cost reductions of approximately 19%, 60%, 11%, and 6% compared to BARON, SPACE4AI-D, PPO_DLX, and minimum k-cut, respectively.

The remainder of this paper is organized as follows. Section 2 introduces the application and the computing continuum model considered in this work. Section 3 presents the formal mathematical formulation of the joint resource allocation and component placement problem. Section 4 provides a high-level overview of the proposed solution framework. The detailed descriptions of the KKT-based optimization approach for resource allocation and the RL method for component placement are elaborated in Sections 5 and 6, respectively. The experimental setup and results are discussed in Section 7, demonstrating the effectiveness of our approach. Related work is reviewed in Section 8, and conclusions are drawn in Section 9.

2 Application and Resource Models

In this section, we introduce the general model developed for our resource optimization and application components placement tool. In particular, we describe the application model in Section 2.1, we discuss the Quality of Service requirements in Section 2.2, and we introduce the computing continuum resource model in Section 2.3.

2.1 Application components model

As in other literature works ([10], [12], [13], [14]), each application is modeled as a DAG (see Figure 1), where the nodes correspond to individual components and the edges represent communication dependencies between them. Figure 1 illustrates a reference application related to the maintenance and inspection of wind farms [4], [10]. The five components which have heterogeneous demands in terms of computational power and QoS constraints, can be executed on different candidate resources. We assume that each component is a Python function that can run in a Docker container deployed in an edge device and in a cloud Virtual Machine (VM). For the sake of simplicity and according to other literature proposals [15], [16], [17], we assume that the DAG includes a single entry point, characterized by the input exogenous workload λ (expressed in terms of requests/sec), and a single exit point. We assume that the inter-arrival time of requests, i.e., $1/\lambda$, is exponentially distributed [18]. Moreover, we consider DAGs including only sequential execution and branches, since, as in [15], we assume that loops are unfolded (or peeled) while parallel execution is not supported in this work.

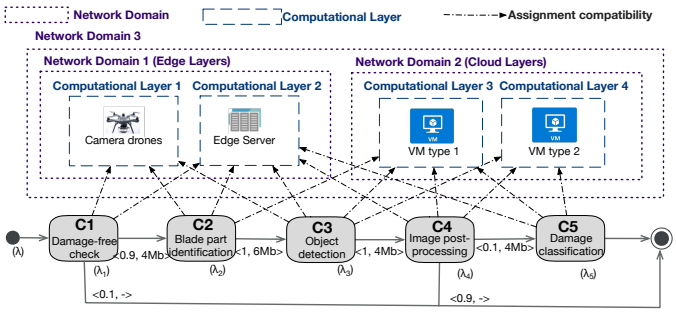


Figure 1: DAG placement, identifying wind turbines blade damage.

We denote the set of all application components by $\mathcal{I}$. The directed edge from node i to node k is labeled with $\langle p^{ik}, \delta^{ik} \rangle$, where p^{ik} is the transition probability from component i to component k, and δ^{ik} denotes the size of data transferred between the components. Each component $i \in \mathcal{I}$ is also characterized by a total load λ_i that depends on λ and on the transition probabilities related to all its predecessors (i.e., all components $k \in \mathcal{I}$ such that $p^{ki} > 0$). In particular, let $Prec^i \subseteq \mathcal{I}$ be the set of all components k executed immediately before i. Then:

$$\lambda_i = \sum_{k \in Prec^i} p^{ki} \lambda_k. \quad (1)$$

2.2 Quality of Service requirements

The main QoS requirement we consider in our system is the response time. For each component $i \in \mathcal{I}$, R^i denotes the average response time.

We define execution paths as sequences of application components that go from the entry point to the exit point

of the DAG. The set of all execution paths ep in the DAG is denoted by $\mathcal{EP}$. We also define a path P as a set of consecutive components included in an execution path $ep \in \mathcal{EP}$. We denote by $\widehat{R}_P$ the response time of each path $P \subseteq ep$.

QoS requirements may be imposed on both the response time of a single component and the response time of all components included in a path. Formally, we define *local constraint* as an upper bound threshold $\overline{LR}_i$ for the response time of an individual component $i \in \mathcal{I}$. We characterize the set of local constraints as:

$$\mathcal{LC} = \{\langle i, \overline{LR}_i \rangle : \ i \in \mathcal{I}, \ R_i \leq \overline{LR}_i\}. \tag{2}$$

Similarly, a *global constraint* is a threshold $\overline{GR}_P$ for the response time of the set of components included in a given path P. The set of global constraints is:

$$\mathcal{GC} = \{\langle P, \overline{GR}_P \rangle : \ P \subseteq ep, \ ep \in \mathcal{EP}, \ \widehat{R}_P \leq \overline{GR}_P\}. \tag{3}$$

Section 3.2 outlines the methodology for determining R_i and $\widehat{R}_P$.

2.3 Resources general model

In our system model, we assume that all resources in computing continuum are virtualized (though our model is flexible enough to accommodate container-based systems), and henceforth, we refer to these resources as virtual machines (VMs). We define distinct computational layers, each comprising multiple instances of a specific VM type. The specific VM type per layer can be selected among multiple candidate VM types during the system design time or determined by the system administrator based on operational requirements [10]. We denote the set of VMs in Edge and Cloud layers by $\mathcal{J}_\mathcal{E} = \{1, ..., E\}$ and $\mathcal{J}_\mathcal{C} = \{E+1, ..., E+C\}$, respectively. Consequently, the set of all VMs in computing continuum will be denoted as $\mathcal{J} = \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$. For each $j \in \mathcal{J}$, the total number of available instances is denoted by $N_j \in \mathbb{N}$. Note that, as each computational layer is associated with a distinct virtual machine type, we henceforth utilize the term "layer j" to denote the virtual machine indexed as j.

Communications in the computing continuum are enabled by a set $\mathcal{D}$ of network domains connecting layers to each other and with the remote cloud back-end. Each $d \in \mathcal{D}$ is associated with a given technology, characterized by the bandwidth B^d. Computational layers are included in, possibly, multiple network domains. For each network domain d, we define $\mathcal{ND}^d = \{l_1, \cdots, l_{n^d}\}$ as the set of layers included in the network domain. To properly evaluate the response time of components, we need to compute the network delay due to data transmissions between consecutive components executed on different layers belonging to the same network domain $d \in \mathcal{D}$, depending on B^d and the amount of transferred data (see, e.g., [19]). Moreover, the network transfer time between consecutive components is needed to correctly evaluate the global execution time of a path, as detailed in Section 3.2.

The cost of virtualized resources, denoted by c_j, is primarily determined in Edge layers by their associated energy consumption, whereas the cost of virtualized resources in Cloud layers is characterized by per-second pricing for VMs [20], [21]. The component placement problem addressed in this paper is illustrated in Figure 1. The objective is to optimally allocate the components of the application DAG across appropriate layers to minimize total costs while satisfying some local and global QoS constraints. All the parameters and variables are summarized in Table 1.

Parameters	
$\mathcal{I}$	Set of component indices
$\mathcal{J}_\mathcal{E}, \mathcal{J}_\mathcal{C}$	Set of indices of all VM types in the Edge and Cloud, respectively.
$\mathcal{J}$	Set of indices of all computing continuum resources.
$\mathcal{D}$	Set of existing network domains
$\mathcal{ND}^d$	Set of layers l included in the network domain d
$\mathcal{LC}$	Set of tuples including a component i and an upper bound threshold for the response time of component i
$\mathcal{P}$	Set of paths in the application DAG
$\mathcal{GC}$	Set of of tuples including a path P and a threshold for the total response time of path P
λ	Input exogenous workload
λ_i	Incoming workload of component i
p^{ik}	Probability of running component k after component i
δ_{ik}	Data size transferred between components i and k
$\tilde{m}^i$	Memory requirement of component i
M_j	Maximum memory available on VM type $j \in \mathcal{J}$
N_j	Number of available instances of VM type $j \in \mathcal{J}$
c_j	Execution cost on VM type $j \in \mathcal{J}$
D_{ij}	Demanding time of component i on layer $j \in \mathcal{J}$
B^d	Bandwidth of network domain d
$\overline{LR}_i$	Threshold for the response time of component i
$\overline{GR}_P$	Threshold for the total response time of a path P
Decision Variables	
x_{ij}	1 if component i placed on layer j, 0 otherwise
n_j	Number of instances on layer j assigned to components

Table 1: Parameters and decision variables

3 Problem Formulation

This section presents the mathematical formulation of our problem. Section 3.1 introduces the decision variables and placement constraints, while the performance model and cost are described in Section 3.2. The complete model is finally presented in Section 3.3.

3.1 Decision variables and allocation constraints

The joint component placement and resource optimization problem we tackle in this paper is formulated as a Mixed Integer Nonlinear Programming (MINLP) problem.

We define the assignment decisions (i.e., we characterize which resource is assigned to each component), by introducing the binary variables:

$$x_{ij} = \begin{cases} 1 & \text{if component } i \text{ is deployed at layer } j \in \mathcal{J}, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Every component can be assigned to a single layer:

$$\sum_{j \in \mathcal{J}} x_{ij} = 1 \qquad \forall i \in \mathcal{I} \tag{5}$$

Moreover, the maximum number of components that can be co-located at each layer j is limited by the available memory M_j. Let $\tilde{m}_i$ be the memory required by component i, as described in Section 2.1. The total memory requirement at layer j can be defined by summing the contribution of all components possibly executed at layer j, which entails:

$$\sum_{i \in \mathcal{I}} \tilde{m}_i x_{ij} \leq M_j \qquad \forall j \in \mathcal{J}, \tag{6}$$

Finally, requests cannot move back from cloud to edge [10]. This means that, if component i is executed on cloud layers, the next consecutive components in application DAG must not be executed on Edge layers. Thus, we impose:

$$p^{ik} x_{ij} \leq \sum_{l \in \mathcal{J}_\mathcal{C}} x_{kl}, \qquad \forall i, k \in \mathcal{I}, \ j \in \mathcal{J}_\mathcal{C}, \tag{7}$$

which ensures that if $p^{ik} x_{ij} = 1$, then component k and all the next consecutive components must also be placed in the cloud.

3.2 Performance Model and cost

This section discusses the performance models that we used to characterize the response time of components executed on different resource types. To predict the performance of each component, we employ M/G/1 queues assuming processor sharing (as in other Edge-Cloud systems research works [10], [19], [22], [23], [24]), which provides a robust framework for modeling systems with general service time distributions and varying workloads. For each layer j, we denote by D_{ij} the demanding time to run component i at layer $j \in \mathcal{J}$, i.e., the average time required to run component i at layer j without resource contention (see [25]). As mentioned in Section 2.3, we denote by N_j the maximum number of instances of VM j that can be selected in the placement (e.g., associated with reserved instances [26], [27], [28], [29]), and we introduce a variable n_j to count the number of instances in use. Note that the load is shared evenly among all VM instances used to run a single component.

As in [10], [19], each VM is modeled as a single server multiple class queue system (i.e., as an individual M/G/1 queue), the response time of a component i possibly deployed at any layer $j \in \mathcal{J}$ can be computed as

$$R_i = \sum_{j \in \mathcal{J}} \frac{D_{ij} x_{ij}}{1 - U_j}, \tag{8}$$

where U_j is the utilization of layer j, defined as

$$U_j = \frac{\sum_{i \in \mathcal{I}} D_{ij} x_{ij} \lambda_i}{n_j}. \tag{9}$$

We further need to prescribe that if any component is executed at a layer j, this is not saturated, i.e., that the equilibrium conditions for the M/G/1 queue hold. In particular, this is equivalent to prescribe

$$\sum_{i \in \mathcal{I}} D_{ij} x_{ij} \lambda_i < n_j. \tag{10}$$

To define the response time of a path comprising consecutive components of the application DAG, it is essential to account for both the response time of each component and the network delay. The network delay incurred by data transmission between two consecutive components when they are placed on different layers (i.e., all components $i, k \in I$ such that $p^{ik} > 0$) is

$$t_{ik} = \frac{\delta_{ik}}{\sum\limits_{j,l \in \mathcal{J}, j \neq l} B_{jl} x_{ij} x_{kl}}, \tag{11}$$

where B_{jl} is the bandwidth between layers j and l. If we consider any path P, i.e., any sequence of components, the relative response time $\widehat{R}_P$ can be defined as follows:

$$\widehat{R}_P = \sum_{i \in P} R_i + \sum_{\substack{i,k \in P, \\ i \in Prec^k}} t_{ik}, \tag{12}$$

where the first term represents the response time of all components in the path, and the last one denotes the network delay due to data transfer operations among different components. The total cost of VMs can be computed as

$$cost = \sum_{j \in \mathcal{J}} c_j n_j. \tag{13}$$

3.3 Optimization problem

The optimization model presented in this work addresses the joint component placement and resource optimization, whose objective is to minimize the total application execution costs while ensuring QoS guarantees. The MINLP formulation of the problem reads:

$$\min_{n_j, x_{ij}} \sum_{j \in \mathcal{J}} c_j n_j \tag{P1a}$$

subject to:

$$\sum_{j \in \mathcal{J}} x_{ij} = 1 \qquad \forall i \in \mathcal{I} \tag{P1b}$$

$$\sum_{i \in \mathcal{I}} \tilde{m}_i x_{ij} \leq M_j \qquad \forall j \in \mathcal{J} \tag{P1c}$$

$$p^{ik} x_{ij} \leq \sum_{l \in \mathcal{J}_\mathcal{C}} x_{kl} \qquad \forall i, k \in \mathcal{I},\ j \in \mathcal{J}_\mathcal{C} \tag{P1d}$$

$$\sum_{i \in \mathcal{I}} D_{ij} x_{ij} \lambda_i < n_j \qquad \forall j \in \mathcal{J} \tag{P1e}$$

$$\sum_{j \in \mathcal{J}} \left(\frac{D_{ij} x_{ij}}{1 - U_j} \right) \leq \overline{LR}_i \qquad \forall \langle i, \overline{LR}_i \rangle \in \mathcal{LC} \tag{P1f}$$

$$\sum_{i \in P} R_i + \sum_{\substack{i,k \in P: \\ i \neq k}} \frac{\delta_{ik}}{\sum\limits_{j,l \in \mathcal{J}, j \neq l} B_{jl} x_{ij} x_{kl}} \leq \overline{GR}_P \qquad \forall \langle P, \overline{GR}_P \rangle \in \mathcal{GC} \tag{P1g}$$

$$n_j \leq N_j \qquad \forall j \in \mathcal{J} \tag{P1h}$$

$$n_j \in \mathbb{N} \qquad \forall j \in \mathcal{J} \tag{P1i}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in \mathcal{I},\ j \in \mathcal{J} \tag{P1j}$$

Constraints (P1b)–(P1e) are already described in previous sections. Constraints (P1f) and (P1g) guarantee the local and global constraints, respectively. Constraint (P1h) ensures that the number of VM instances at layer j assigned to the components does not exceed the number of available VM instances at layer j. Finally, Constraints (P1i)–(P1j) define the decision variables domain. Note that, while our framework minimizes cost as the primary objective, latency is rigorously enforced through strict response time constraints, ensuring support for latency-sensitive applications within predefined performance bounds.

Theorem 3.1. *Constraints* (P1g) *in problem* (P1) *are non-convex.*

Proof. The proof is given in Appendix. □

4 General Solution

Problem (P1), discussed in the preceding section, is a MINLP with non-convex constraints (P1g). Non-convex problems present substantial challenges due the absence of optimality guarantees and their classification as NP-hard problems. To simplify (P1), we decompose it into two subproblems, each focusing on one of the decision variables: either x_{ij} or n_j.

In the first subproblem, we assume a fixed component assignment, thereby setting a priori the decision variables x_{ij}. By doing so, the transmission delay in (P1g) becomes known, leading to a convex formulation of (P1g). Specifically, the inequality (P1g) can be rewritten as:

$$\sum_{i \in P} R_i \leq TH_P$$

where

$$TH_P = \overline{GR}_P - \sum_{\substack{i,k \in P: \\ i \neq k, \\ i \in Prec^k}} \frac{\delta_{ik}}{\sum\limits_{j,l \in \mathcal{J}, j \neq l} B_{jl} x_{ij} x_{kl}} \tag{14}$$

is the new threshold for the consecutive components in path P, which takes into account the transmission delay. So, for every execution path we can define $\langle P, TH_P \rangle \in \mathcal{GC}$ as an equivalent global constraint.

In this setting, the problem can be formulated as follows:

$$\min_{n_j} \sum_{j \in \mathcal{J}} c_j n_j$$

subject to:

$$\sum_{i\in\mathcal{I}} D_{ij}x_{ij}\lambda_i < n_j \qquad \forall j \in \mathcal{J}$$

$$\sum_{j\in\mathcal{J}} \frac{n_j D_{ij}x_{ij}}{n_j - \sum_{i'\in\mathcal{I}} D_{i'j}x_{i'j}\lambda_{i'}} \le \overline{LR}_i \qquad \forall \langle i, \overline{LR}_i\rangle \in \mathcal{LC}$$

$$\sum_{i\in P}\sum_{j\in\mathcal{J}} \frac{n_j D_{ij}x_{ij}}{n_j - \sum_{i'\in\mathcal{I}} D_{i'j}x_{i'j}\lambda_{i'}} \le TH_P \qquad \forall \langle P, TH_P\rangle \in \mathcal{GC}$$

$$n_j \le N_j \qquad \forall\, j \in \mathcal{J}$$

$$n_j \in \mathbb{N} \qquad \forall\, j \in \mathcal{J}$$

If, as in [30], [31], we relax the integrality of n_j and define the constants $z_{ij} = D_{ij}x_{ij}$, $Z_{Pj} = \sum_{k\in P} z_{kj}$ and $L_j = \sum_{i\in\mathcal{I}} D_{ij}x_{ij}\lambda_i$, then we can simplify the optimization problem as follows:

$$\min_{n_j} \sum_{j\in\mathcal{J}} c_j n_j \qquad \text{(P2a)}$$

subject to:

$$\sum_{j\in\mathcal{J}} \frac{n_j z_{ij}}{n_j - L_j} \le \overline{LR}_i \qquad \forall \langle i, \overline{LR}_i\rangle \in \mathcal{LC} \qquad \text{(P2b)}$$

$$\sum_{j\in\mathcal{J}} \frac{n_j Z_{Pj}}{n_j - L_j} \le TH_P \qquad \forall \langle P, TH_P\rangle \in \mathcal{GC} \qquad \text{(P2c)}$$

$$n_j \le N_j \qquad \forall j \in \mathcal{J} \qquad \text{(P2d)}$$

$$n_j > L_j \qquad \forall j \in \mathcal{J} \qquad \text{(P2e)}$$

Theorem 4.1. *Problem* (P2) *is convex.*

Proof. The proof is given in Appendix. □

Although problem (P2) is convex, the constraints are coupled, meaning the solution for n_j is interdependent across different constraints. The global and local constraints involve sums over all j, making it challenging to determine the optimal values of n_j in closed form. Consequently, we resort to Hierarchical Decomposition [32] to approximate the nonlinear problem by solving a series of subproblems hierarchically. In Section 5, we propose a KKT-based approach to find a near-optimal solution for the problem.

The second subproblem of (P1) pertains to the component placement problem, where x_{ij} are the only decision variables. For each value of x_{ij}, we can obtain a near-optimal solution for n_j by solving the first subproblem. To address this second subproblem, we proposed a reinforcement learning approach, as outlined in Section 6.

An overview of the general solution is presented in Figure 2, illustrating the iterative training process of the proposed RL framework. The process begins with a random initial placement (however compliant with (P1) constraints), and at each iteration, the analytical method described in Section 5 is used to compute the optimal number of VM instances, $\boldsymbol{n}^*$, for the given placement. This enables an efficient evaluation of the system performance and cost. The resulting performance metrics are then used to derive a reward signal, which guides the RL agent in updating its placement strategy. This loop continues until the learning process reaches the maximum number of iterations. The integration of closed-form analytical solutions into the RL loop not only ensures scalability but also accelerates training by eliminating the need for computationally intensive simulations or external solvers.

5 Resource Allocation Problem Solution through KKT Conditions

In this section, we propose an approach to solve the resource allocation problem (P2). As discussed in the previous section, the interdependencies between the decision variables n_j across both local and global constraints make it infeasible to derive a general closed-form solution for n_j. This complexity arises due to the collocation of components on shared resources and the interaction between local and global constraints. Therefore, to address this challenge, we employ a hierarchical decomposition approach to find a near-optimal solution for n_j. Our approach involves first solving (P2) under local constraints independently—disregarding the global constraints—to obtain the optimal number of resources that satisfy the local conditions (see Section 5.1). This solution serves as a lower bound for (P2). Next, we consider the complete (P2) formulation that includes the global constraints to refine the resource allocation and ensure that the global requirements are met (see Section 5.2).

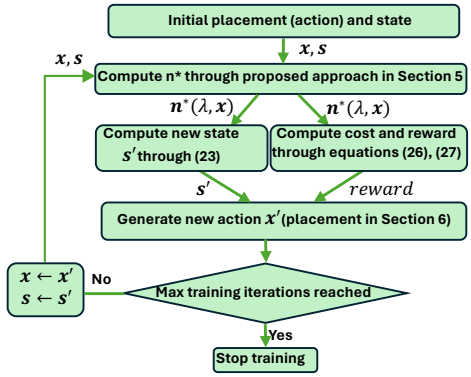


Figure 2: General solution flowchart.

5.1 Solving (P2) without global constraints

Given that local constraints are independent, the optimal number of instances can be determined by equating the response time of component i to the local constraint threshold $\overline{LR}_i$. This approach ensures the minimum number of VM instances required to satisfy $\overline{LR}_i$, defined as n_j^i, is computed efficiently, because if the constraint holds as strict inequality rather than equality, it indicates an over-allocation of resources. Specifically, if a single component i is allocated at layer j, then we impose:

$$\frac{n_j^i z_{ij}}{n_j^i - L_j} = \overline{LR}_i,$$

and we get

$$n_j^i = \frac{L_j \overline{LR}_i}{\overline{LR}_i - z_{ij}}. \qquad (15)$$

In the case that more than one components are running on the same layer j (i.e. $\sum_{i\in\mathcal{I}} x_{ij} > 1$) and a local constraint is defined for multiple components, we will have:

$$n_j = \max_{\langle i, \overline{LR}_i\rangle \in \mathcal{LC}} n_j^i. \qquad (16)$$

Since n_j must be integer, we round n_j into the next integer, and we define $NC_j = \lceil n_j \rceil$ as the minimum number of resource j to satisfy local constraints. Moreover, if resource j is not running any components with local constraint, we only need to avoid saturation by considering $n_j > L_j$, thus $NC_j = \lfloor L_j \rfloor + 1$[1].

5.2 Solving (P2) with global constraints

To identify the optimal n_j that simultaneously satisfy the global constraints, we must introduce additional virtual

1. If L_j is an integer, $\lceil L_j \rceil$ does not guarantee the saturation constraint $n_j > L_j$, thus we define $NC_j = \lfloor L_j \rfloor + 1$ to ensure that the constraint holds.

machines n'_j that are necessary in addition to NC_j. So we define n_j as follows:

$$n_j = NC_j + n'_j$$

Moreover, according to the constraint on total number of resources (P1h) in the complete problem, we define an upper bound for n'_j as follows:

$$N'_j = N_j - NC_j, \tag{17}$$

where $N'_j \geq 0$.

Since solving (P2) with multiple global constraints is challenging due to the interdependencies of n'_j across different global constraints, we will consider global constraints incrementally. The closed-form expressions derived for local constraints (i.e., NC_j in the preceding subsection) serve as the initial lower bound for the global constraints. We solve (P2) with one global constraint at a time, iteratively updating the lower bound at each layer based on the solution of the current global constraint. This approach enables incremental addressing of subsequent global constraints until all global constraints are considered, resulting in a feasible and near-optimal solution.

5.2.1 Solving (P2) with one global constraint

In the following, we consider (P2) with only one global constraint $\langle P, TH_P \rangle \in \mathcal{GC}$. Let $\mathcal{J}_P \subseteq \mathcal{J}$ be the set of all the resources running the components involved in the global constraint. Consequently, with the removal of the local and saturation constraints, problem (P2) can then be expressed as follows:

$$\min_{n'_j} \sum_{j \in \mathcal{J}_P} c_j n'_j \tag{P3a}$$

subject to

$$\sum_{j \in \mathcal{J}_P} \frac{(NC_j + n'_j) Z_{Pj}}{NC_j + n'_j - L_j} \leq TH_P \tag{P3b}$$

$$0 \leq n'_j \leq N'_j \qquad \forall j \in \mathcal{J}_P \tag{P3c}$$

Theorem 5.1. *The optimal solution n'^* of problem* (P3) *is given as follows:*

If $\sum_{j \in \mathcal{J}_P} \frac{(NC_j + N'_j) Z_{Pj}}{NC_j + N'_j - L_j} > TH_P$, *then* (P3) *is unfeasible.*

If $\sum_{j \in \mathcal{J}_P} \frac{NC_j Z_{Pj}}{NC_j - L_j} \leq TH_P$, *then* $n'^*_j = 0$ *for any* $j \in \mathcal{J}_P$.

If $\sum_{j \in \mathcal{J}_P} \frac{(NC_j + N'_j) Z_{Pj}}{NC_j + N'_j - L_j} \leq TH_P < \sum_{j \in \mathcal{J}_P} \frac{NC_j Z_{Pj}}{NC_j - L_j}$, *then*

$$n'^*_j = \begin{cases} N'_j & \forall\, j \in \mathcal{J}_P^{=}, \\ L_j - NC_j + \alpha \sqrt{\frac{L_j Z_{Pj}}{c_j}} & \forall\, j \in \mathcal{J}_P^{>}, \\ 0 & \forall\, j \in \mathcal{J}_P^{0}, \end{cases} \tag{18}$$

where $\{\mathcal{J}_P^0, \mathcal{J}_P^>, \mathcal{J}_P^=\}$ *is a partition of* $\mathcal{J}_P$,

$$\alpha := \frac{\sum_{j \in \mathcal{J}_P^>} \sqrt{L_j Z_{Pj} c_j}}{TH_P - \sum_{j \in \mathcal{J}_P^>} Z_{Pj} - \sum_{j \in \mathcal{J}_P^0} \frac{NC_j Z_{Pj}}{NC_j - L_j} - \sum_{j \in \mathcal{J}_P^=} \frac{(NC_j + N'_j) Z_{Pj}}{NC_j + N'_j - L_j}}, \tag{19}$$

and the following conditions hold:

$$\alpha \leq \beta_j \qquad \forall\, j \in \mathcal{J}_P^0, \tag{20}$$

$$\alpha \geq \gamma_j \qquad \forall\, j \in \mathcal{J}_P^=, \tag{21}$$

$$\beta_j < \alpha < \gamma_j \qquad \forall\, j \in \mathcal{J}_P^>, \tag{22}$$

with

$$\beta_j := \frac{NC_j - L_j}{\sqrt{L_j Z_{Pj} / c_j}}, \qquad \gamma_j := \frac{NC_j + N'_j - L_j}{\sqrt{L_j Z_{Pj} / c_j}}.$$

Proof. The proof is given in Appendix. □

Theorem 5.1 investigates all possible conditions related to the global constraint $\langle P, TH_P \rangle$. The first condition states that if all available instances across all layers involved in the path are allocated and the constraint remains unsatisfied, no feasible solution exists for the problem. The second condition asserts that no additional instances are required if the current allocation satisfies the constraint. Finally, the third and most significant condition states that if the current allocation fails to satisfy the constraint, yet full allocation of instances would meet it, the optimal number of additional instances across all layers can be determined using the closed-form expression in (18), provided that conditions (20)-(22) are met. In other words, Theorem 5.1 states that if (P2) is feasible and a specific resource partitioning $\{\mathcal{J}_P^0, \mathcal{J}_P^>, \mathcal{J}_P^=\}$ satisfies conditions (20)-(22), then a closed-form solution exists for determining the optimal additional instances. Although determining the optimal resource partitioning is inherently a combinatorial problem, Theorem 5.1 establishes well-defined criteria (β_j and γ_j) for sorting resources, enabling the efficient computation of the optimal partitioning and, consequently, the optimal value of $\boldsymbol{n}^*$. Furthermore, since α is derived from an analytical formula rather than requiring exhaustive combinatorial enumeration, our following exact algorithm with polynomial time complexity effectively transforms the problem into a more computationally tractable form, significantly reducing its complexity (see Algorithm 1). First, we sort resources in $\mathcal{J}_P$ based on the β_j and γ_j values. We define the list of layer indices $\mathcal{J}_{P_\beta} = [j_1, \ldots, j_{|\mathcal{J}_P|}]$ which is sorted as $\beta_{j_1} \leq \beta_{j_2} \leq \cdots \leq \beta_{j_{|\mathcal{J}_P|}}$, and define $\mathcal{J}_{P_\gamma} = [j_1, \ldots, j_{|\mathcal{J}_P|}]$ which is sorted according to $\gamma_{j_1} \leq \gamma_{j_2} \leq \cdots \leq \gamma_{j_{|\mathcal{J}_P|}}$. The parameters β_j and γ_j capture the relative availability and cost-effectiveness of resource j, considering both its current allocation and potential expansion, respectively. Specifically, β_j reflects the extent to which the already allocated instances (NC_j) deviate from the required L_j, normalized by cost and demand factors. Since β_j is tied to existing allocations, the impact of cost c_j on the optimal solution is relatively limited. In contrast, γ_j represents the deviation of the fully allocated resource j ($NC_j + N'_j$) from L_j, indicating the cost-effectiveness of expanding resource j with additional instances. A higher γ_j implies a higher cost for fully allocating that resource, making it less desirable in an optimal allocation. Based on this reasoning, the algorithm prioritizes selecting resources with smaller γ_j to minimize cost when fully allocating new instances (N'_j). At the same time, it prefers resources with larger β_j, ensuring that resources with higher available capacity (NC_j) are utilized first to preserve feasibility and avoid relying on more expensive expansions. This resource partitioning strategy balances cost efficiency and resource availability, optimizing the allocation process.

Algorithm 1 Finding the optimal partition $\{\mathcal{J}_P^0, \mathcal{J}_P^>, \mathcal{J}_P^=\}$ for (P3)

```
for h = 0, ..., |J_P| do
    J_P^= ← {j_i ∈ J_{P_γ} : 1 ≤ i ≤ h}
    for k = 0 ..., |J_P| do
        J_P^0 ← {j_i ∈ J_{P_β} : |J_P| − k + 1 ≤ i ≤ |J_P|} \ J_P^=
        J_P^> ← {j_i ∈ J_{P_β} : 1 ≤ i ≤ |J_P| − k} \ J_P^=
        compute α defined in (19)
        if conditions (20)–(22) hold then STOP
        end if
    end for
end for
```

Given the properties of the optimal solution of problem (P3), proved in Theorem 5.1, Algorithm 1 terminates after at most $O(|\mathcal{J}_P|^2)$ iterations providing the optimal partition $\{\mathcal{J}_P^0, \mathcal{J}_P^>, \mathcal{J}_P^=\}$ (see the Appendix for some additional considerations). Note that, in the worst case, during the final iteration of the outer loop, all layers of $\mathcal{J}_P$ will be assigned to the equality set ($\mathcal{J}_P^= \leftarrow \mathcal{J}_P$, as stated in line 2). This implies that all remaining VM instances (N'_j) will be utilized across all layers.

5.2.2 Solving (P2) with multiple global constraints

After solving the problem for a given global constraint $\langle P, TH_P\rangle$, the lower bound for each layer is updated, and the remaining global constraints are addressed incrementally. The approach involves adding one constraint at a time, where the solution of each constraint helps identify additional lower bounds. Once all global constraints have been addressed, the process yields the final solution.

Algorithm 2 outlines the proposed approach in detail for obtaining a near-optimal number of resources. We first obtain the required number of instances to satisfy local constraint as described in Section 5.1 (lines 1-6). For resources running at least one component ($L_j > 0$) but not involved in any local constraints, we compute the necessary number of instances to prevent saturation (lines 7-10). Next, for all resources, we check if the required number of instances (NC) exceeds the maximum available instances; if so, the solution is deemed unfeasible (lines 11–14), otherwise, we add the global constraints iteratively. If the total instances of all resources involved in the global constraint $\langle P, TH_P\rangle$ are insufficient to meet the threshold TH_P (first condition of Theorem 5.1), then the placement is also unfeasible (lines 15-19). If the currently assigned number of instances (NC) is sufficient to satisfy $\langle P, TH_P\rangle$, no additional resources are required (lines 20-21), otherwise we first obtain the subsets $\mathcal{J}_P^0$, $\mathcal{J}_P^>$ and $\mathcal{J}_P^=$ through Algorithm 1 to compute α and n'^*_j and then update NC_j for all resources involved in path P (lines 22-28). Finally, the Algorithm returns a tuple, which includes the feasibility flag and the required number of instances for all resources (lines 29-30). According to the complexity of Algorithm 1, the time complexity of Algorithm 2 to obtain a near optimal number of resources is at most $O(|\mathcal{GC}| \times |\mathcal{J}_P|^2)$.

Algorithm 2 Heuristic algorithm for solving (P2)

1: **Input:** $\mathcal{J}$, $\mathcal{I}$, z, Z, L, N, c, DAG, $\mathcal{LC}$, $\mathcal{GC}$
2: **Initialization:** $NC_j = 0, n^*_j = 0 \quad \forall j \in \mathcal{J}$, $feasible \leftarrow True$
3: **for** $\langle i, \overline{LR}_i\rangle \in \mathcal{LC}$ **do**
4: Compute n_j using (15) and (16)
5: $NC_j = \lceil n_j \rceil$
6: **end for**
7: **for** $j \in \mathcal{J}$ **do**
8: **if** $NC_j = 0$ and $L_j > 0$ **then**
9: $NC_j = \lceil L_j \rceil$
10: **end if**
11: **if** $NC_j > N_j$ **then**
12: $feasible \leftarrow False$
13: **end if**
14: **end for**
15: **if** $feasible$ **then**
16: **for** $\langle P, TH_P\rangle \in \mathcal{GC}$ **do**
17: **if** $\sum_{j \in \mathcal{J}_P} \frac{N_j Z_{Pj}}{N_j - L_j} > TH_P$ **then**
18: $feasible \leftarrow False$
19: Break;
20: **else if** $\sum_{j \in \mathcal{J}_P} \frac{NC_j Z_{Pj}}{NC_j - L_j} \leq TH_P$ **then**
21: $n'^*_j = 0$
22: **else**
23: Find the partition $\{\mathcal{J}_P^0, \mathcal{J}_P^>, \mathcal{J}_P^=\}$ from Algorithm 1
24: Compute n'^*_j through formula (18)
25: **end if**
26: $NC_j \leftarrow NC_j + n'^*_j \quad \forall j \in \mathcal{J}_P$
27: **end for**
28: **end if**
29: $\boldsymbol{n}^* \leftarrow \{NC_j | j \in \mathcal{J}\}$
30: **Return** $(feasible, \boldsymbol{n}^*)$

6 Component placement through Reinforcement Learning

In the previous sections, we assumed a fixed component placement to simplify the problem and achieve a near-optimal allocation of resources. In this section, we address the component placement problem, where the objective is to find a near-optimal solution for the decision variable x_{ij}. In Section 6.1, we model the component placement problem as a Markov decision process (MDP). Subsequently, we introduce a technique to address scalability challenges in large-scale systems, along with its application to our problem in Section 6.2.

6.1 Component placement as an MDP

We formulate the component placement problem as a discrete-time infinite-horizon MDP, defined by a 5-tuple $\langle \mathbf{S}, \mathbf{A}, P, c, \gamma\rangle$. Here, $\mathbf{S}$ represents the (potentially infinite) set of all possible system states, and $\mathbf{A}(s)$ denotes the finite set of actions available in state s. $P(s'|s,a)$ represents the transition probability from state s to a state s' given an action $a \in \mathbf{A}(s)$. The immediate cost incurred when the system transitions from state s to state s' as a result of action a is denoted by $c(s,a,s')$. Finally, $\gamma \in [0,1]$ is the discount factor, which governs the weight assigned to future costs. We define the agent state as

$$s = (\lambda,\ \mathbf{U}, \mathbf{x}), \tag{23}$$

where $\mathbf{x}$ is the component to layer assignment matrix while $\mathbf{U}$ is a vector of size $|\mathcal{J}|$ where each element corresponds to a specific layer and represents the utilization of that layer based on the optimal number of required instances per layer ($\boldsymbol{n}^*(\lambda, \mathbf{x})$) determined in Section 5. λ is an exogenous variable beyond the agent's control, i.e., its variability is independent of the actions taken and is observed directly from the environment. In contrast, $\mathbf{U}$ is influenced by the selected actions and hence is a function of $\boldsymbol{n}^*$.

An action involves selecting a placement. However, defining the placement poses a significant challenge, as it directly impacts the action space and, consequently, the performance of the RL approach. To address this, we systematically explore all possible action definitions for the component placement problem and categorize them into two approaches: (i) the *partial solution*, where a single component is migrated to a specific resource per action, and (ii) the *complete solution*, where the placement of all components is determined in a single action. Notably, in the complete solution approach, a more compact state representation can be employed, as the decision variable $\mathbf{x}$ can be omitted from the state (see equation (23)), given that it is directly determined by the action. Accordingly, the possible action definitions are as follows:

- **Single migration** *(partial solution):* A straightforward approach to defining an action for the placement problem is to represent it as a pair of a component and a resource, (C_i, res_j), indicating that component C_i migrates to re-

source j. Specifically, with $|\mathcal{I}|$ components and J_i compatible resources for each component, the total number of possible actions becomes $\sum_{i=1}^{|\mathcal{I}|} J_i$. Note that, in some states, the agent may choose not to change the placement selected in the previous time window; we model this scenario by introducing an action η that represents the *do nothing* choice. The set of all the possible actions $\mathbf{A}(s)$ is hence given by

$$\mathbf{A}(s) = \left\{ a_0, a_1, \ldots, a_{\,|\sum_{i=1}^{|\mathcal{I}|} J_i|-1} \right\} \cup \{\eta\}, \tag{24}$$

where each a_k represents the action of raising to 1 the variable x_{ij}, where $i = max\{m|\sum_{i=1}^{m} J_i \leq k\}$ and $j = k - \sum_{i=1}^{m} J_i - 1$. Figure 3 (top row) illustrates the actions of a system comprising 5 components and 3 resources. This definition is advantageous in terms of action space size, as the space grows linearly with the number of components and resources. However, a major challenge arises: this action definition does not provide a complete solution to the problem at each step, making the solution for the current step (at a given λ) heavily influenced by its past actions which can lead to instability because consecutive states and rewards are not independent. As a result, the agent might struggle[2] to learn how to effectively handle the system dynamics and find a near-optimal placement for a specific λ.

- **Discrete action** *(complete solution):* To address the issue of partial solutions mentioned earlier, we can assign an integer to each possible placement, representing a complete solution, and define the action as a discrete integer. This approach ensures that the action at each step is independent of previous steps, allowing the agent to efficiently learn the optimal placement based on the system dynamics (λ). However, a major challenge with this method is the exponential growth of the action space. Specifically, with $|\mathcal{I}|$ components and J_i compatible resources for each component, the total number of possible actions becomes $\prod_{i=1}^{|\mathcal{I}|} J_i$. By introducing an action η that represents the *do nothing* choice, the set of all the possible actions $\mathbf{A}(s)$ is hence given by

$$\mathbf{A}(s) = \left\{ a_0, a_1, \ldots, a_{\,|\prod_{i=1}^{|\mathcal{I}|} J_i|-1} \right\} \cup \{\eta\}, \tag{25}$$

where each a_k represents a complete solution for binary variable $x_{ij}\ \forall i \in \mathcal{I}, \forall j \in \mathcal{J}$ Although this definition resolves the issue of action dependency, it becomes impractical for large-scale systems, as the action space quickly expands to trillions of actions.

- **Multi-discrete action** *(complete solution):* To address the scalability issue described above, we employ a technique known as *action branching* which utilizes a multi-discrete action representation. The details of this technique, along with its application to our problem, are provided in Section 6.2.

Accordingly, in the implementation, actions correspond to placements represented by the vector $\mathbf{x}$—a $|\mathcal{I}|$-dimensional array where each index corresponds to a component, and the value indicates the index of the resource to which it is assigned. The agent state is a dictionary containing the workload λ (a float normalized to $[0, 1]$) and the utilization vector $\mathbf{U}$.

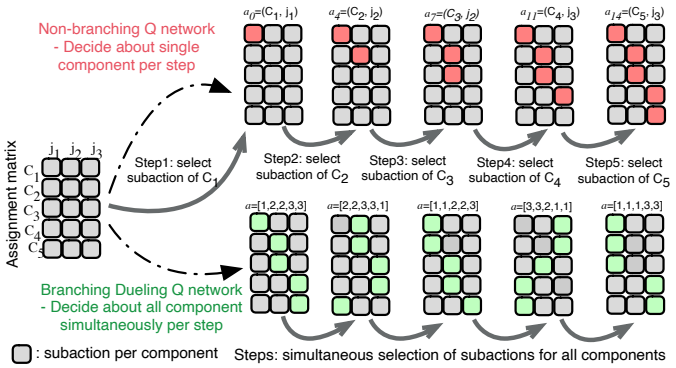


Figure 3: Action definition, single migration vs multi-discrete action.

We associate a cost $c(s, a, s')$ to each triple state-action-next state, normalized between 0 and 1, which reflects the impact of its actions on various aspects of the environment state. The primary factor considered is the infeasibility of the solution, as this represents the most critical outcome. A placement is deemed infeasible if the maximum number of resources assigned to the components is insufficient to meet the QoS constraints (see the first condition in Theorem 5.1). This is represented by the boolean variable $feasible$, as derived from the resource optimization problem outlined in Section 5 (see the output of Algorithm 2). Furthermore, according to the constraints in the general problem (P1), a placement is considered infeasible if the memory requirements of components are not met (constraint (P1c)) or if requests are redirected from the cloud back to the edge, violating constraint (P1d). Therefore, we assign the maximum possible cost to any infeasible placement, as the agent must not prioritize reducing operational costs at the expense of feasibility. For feasible placements, the overall cost is determined by the execution cost, which is calculated based on the assigned resources and the optimal number of instances and normalized with respect to the highest execution cost denoted as C^{max}, which occurs when all resources are fully allocated. Therefore, the cost $c(s, a, s')$ is given by

$$c(s,\ a,\ s') = \begin{cases} 1, & \text{if constraint (P1d) or (P1c) are violated} \\ & \text{OR } feasible \text{ is } False \\ \frac{\sum_{j \in \mathcal{J}} c_j n_j^*}{C^{max}} * \beta & \text{otherwise,} \end{cases} \tag{26}$$

where β is a scaling parameter to create a clear distinction for the agent between the maximum execution cost and the infeasibility cost. Finally, we define the agent reward as

$$reward = 1 - c(s,\ a,\ s'). \tag{27}$$

6.2 Action branching for component placement

In traditional RL methods, the policy or value function usually selects a single action from a large, discrete or continuous action space. However, when the action space grows, the complexity and memory requirements of the policy increase significantly. Action branching [33] is an RL technique designed to efficiently manage large or multi-dimensional action spaces by dividing them into smaller sub-spaces, allowing for separate, specialized decision-making in each sub-space while still coordinating those decisions for the overall action selection. Instead of selecting a single action from a vast space, action branching uses multiple *branches*—smaller, independent networks that handle specific parts of the action space (action-dimension). These

2. In an early implementation of the proposed framework with this action setup, the agent could not converge even considering small systems and performing a significantly large number of steps.

branches are coordinated by a central *shared network*, that processes environmental inputs and provides shared feature representations of the environment state to each branch. The final action is generated by combining the outputs of all branches. Since each branch specializes in a part of the action space, the overall action is a vector or composition of these smaller, independently learned actions. The motivation behind action branching is to reduce computational complexity, scale efficiently to large action spaces, enable independent learning for each subspace, and facilitate parallel decision-making. A prominent architecture utilizing this approach is the Branching Dueling Q-Network (BDQ) [33], an extension of the Dueling DQN [34].

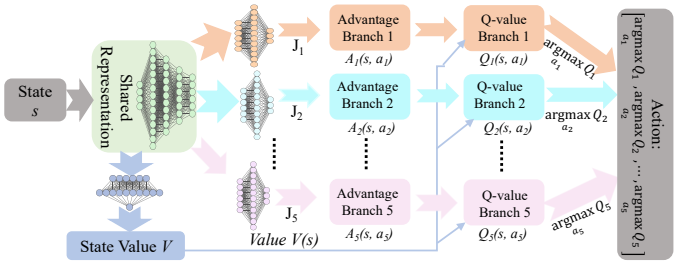


Figure 4: BDQ network architecture, for 5 components and 3 resources.

As shown in Figure 4, the input state s (see equation (23)) outputs a shared network representation through feature processing. Then, the value branch maps from the shared representation to the value $V(s)$, while multiple advantage branches (corresponding to the total number of components in our study) map from the shared representation to the advantages $A_i(s, a_i)$ for individual subactions. The subaction a_i of branch i belongs to $\mathcal{A}_i$ and the dimension $|\mathcal{A}_i|$ of $\mathcal{A}_i$ is J. Each advantage branch i independently calculates its Q-values $Q_i(s, a_i)$ as follows:

$$Q_i(s, a_i) = V(s) + \left(A_i(s, a_i) - \frac{1}{|\mathcal{A}_i|} \sum_{a'_i \in A_i} A_i(s, a'_i) \right).$$

The selection of subsections within each branch is guided by the Q values and an $\epsilon - greedy$ strategy.

As discussed in Section 6.1, defining a complete solution for each individual action leads to an exponential increase in the action space as the number of components and compatible resources grows. However, since the placement of each component can be decided independently and in parallel, we can leverage the *action branching* technique to significantly reduce the action space. To achieve this, we define the placement of each component as an independent sub-action (branch), with its compatible resources as the action dimension. Consequently, the agent action is modeled as a multi-discrete action (rather than a discrete one), represented as a list with a length equal to the number of components (see Figure 3 bottom row). Each list item corresponds to a sub-action, where the agent selects among the compatible resources for that specific component. By adopting this approach, the output size of the network, which would be exponential in the case of complete solutions ($\prod_{i=1}^{|\mathcal{I}|} J_i$, see *discrete action* in Section 6.1), is reduced to a linear size of $O(|\mathcal{I}| \times \max_{i \in \mathcal{I}} J_i)$.

7 Experimental Results

To assess the performance and scalability of our framework, we conducted a comparative analysis with BARON [9], a global solver, and two state-of-the-art approaches proposed in [10] and [11]. Additionally, we evaluated our framework against a baseline method based on the Ford-Fulkerson algorithm for solving the minimum k-cut problem. We trained the RL agent by relying on the RLlib [35] library and employed both the standard Proximal Policy Optimization (PPO) [36] and an enhanced version of PPO incorporating the action branching technique, referred to as *PPO_AB* (see Section 6.2). The experimental setup is described in Section 7.1. Section 7.2 presents the scalability analysis and performance evaluation of the resource optimization approach under a fixed placement scenario, benchmarking it against the BARON solver. Finally, Section 7.3 evaluates our framework for solving the complete problem, which encompasses both resource optimization and component placement, exploring both discrete and multi-discrete action spaces to assess their effectiveness in component placement, compared against BARON solver, the minimum k-cut-based baseline method, as well as the solutions proposed in [10] and [11]. To quantitatively compare an alternative method with our approach, we define the *Cost ratio* as

$$\text{Cost ratio} = \frac{(OtherMethodCost - ProposedMethodCost)}{ProposedMethodCost} \times 100. \tag{28}$$

Positive values indicate that $OtherMethod$ incurs higher costs relative our $ProposedMethod$.

A detailed statistical analysis of the cost differences, including significance testing with robust ANCOVA [37], is provided in Appendix.

7.1 Experimental setup

In our experiments, we randomly generated a large set of instances, which, nevertheless, are representative of real systems. Similarly to [5] [10] and [38], we evaluate the proposed framework under three scenarios of varying scales, specifically application DAGs comprising 7, 10, and 15 components. Additionally, a small-scale scenario with 5 components is analyzed to assess the performance of the full framework, as detailed in Section 7.3. We report the average *Cost ratio* and execution time achieved by considering 5 random instances with the same size. As in [38], we randomly generated the application DAG for every instance by relying on Networkx. We also considered at most 5 computational layers in Edge and Cloud. The cost of resources are set based on AWS pricing in [20] and the component demands are set, inspired from [39], randomly in the range of [0.1, 0.5] seconds and proportional to the associated costs. We introduced up to 7 local and global constraints. The thresholds for local and global constraints are established based on the demands, with constraints assigned a threshold value of 2.5 times the corresponding demand [15]. To train the RL agent in Section 7.3, we define each episode to span 360 steps, corresponding to a one-hour simulation, with a control time period of 10 seconds per step. The workload follows a bimodal distribution, exhibiting two peaks per episode— a pattern commonly observed in real-world systems, making the simulation more representative of practical scenarios [40]. Additional details of scalability analysis

and the parameters adopted are presented in Table 3 and Table 4, respectively, in Appendix. The network parameters in Table 3 are used for PPO and the shared network of PPO_AB, where each branch of PPO_AB consists of a single layer with 128 neurons and ReLU as activation function, across all scenarios. All the results presented for PPO and PPO_AB pertain to the evaluation phase of the RL agents. The code used to generate the random instances and run our experiments are available in Zenodo[3].

7.2 Resource Optimization Performance Evaluation

To validate the KKT-based approach to determine the optimal resource allocation discussed in Section 5, we conduct a comparative analysis with the BARON solver [9]. Given that the problem (P2) possesses a linear objective function and incorporates convex constraints (x_{ij} remains a fixed parameter, while n_j serves as the sole decision variable), a global MINLP solver such as BARON is usually able to identify the optimal global solution through interval analysis and range reduction techniques within a branch-and-bound framework. Since the KKT-based approach works based on closed form equations (15), (16) and (18) in Theorem 5.1, we refer to this approach as CF (short for Closed Form) in the figures presented in this subsection. Note that, the general problem (P1) is computationally challenging for BARON (and other mixed-integer nonlinear solvers). A comparison between BARON for solving the complete problem and our full framework is provided in Section 7.3.1. Figure 5 presents a comparison between the CF (proposed method) and BARON for a single representative instance in a small system composed of four components and two resources. In this configuration, components $C1$ and $C3$ are assigned to $VM1$, while components $C2$ and $C4$ are allocated to $VM2$. Figure 5a illustrates the difference between the optimal number of VM instances determined by CF and those identified by BARON. The results demonstrate that CF and BARON effectively allocate the VMs, ensuring an optimal and equal distribution. Occasionally, they interchangeably utilize VMs 1 and 2, resulting in a maximum discrepancy of only one VM between CF and BARON. This slight variation often arises because CF rounds the continuous variable n_j to the next integer for all resources (as described in Section 5), whereas BARON employs more advanced techniques to compute the exact optimal number of instances. However, the difference of one instance out of many instances (up to 200 and 400 for VM1 and VM2, respectively) is negligible as the load and the layer size increase. Figure 5b emphasizes this fact by showing that the cost ratio between CF and BARON is mostly zero and only occasionally rises above zero, with a decreasing trend as the workload (λ) increases. This is due to the growing number of VM instances required to meet the system constraints as λ increases, while the difference still remains at most one instance, which results in a reduction in the cost ratio. Figures 5c and 5d illustrate the response times (RT) of components and paths, along with their corresponding thresholds, for CF and BARON, respectively. The results exhibit nearly identical behavior, with response times being virtually equal between the two methods.

3. https://doi.org/10.5281/zenodo.15037926

Figure 6 presents the large-scale analysis averaging across 5 random instances, which follows a similar trend to the small-scale system. The comparison of solution time and cost ratio between CF and BARON reveals that CF can achieve comparable solutions and costs at least 10 times faster (see Figure 6b).

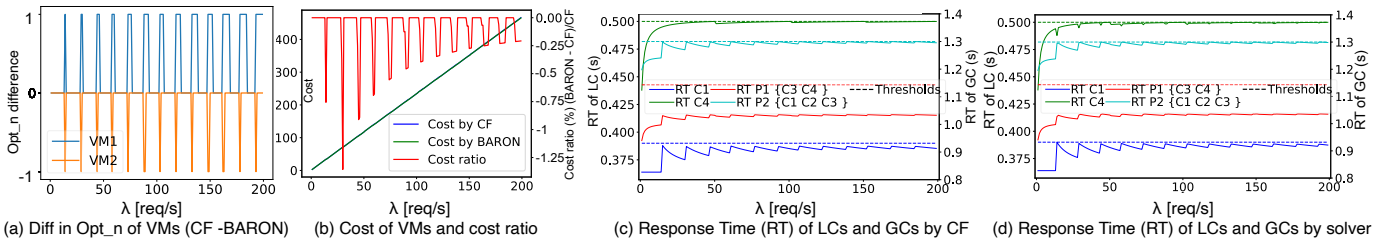
7.3 Overall Problem Performance Evaluation

This section evaluates our full framework (which solves the complete problem in (P1)) with several approaches. The evaluation spans systems of varying scales, from small-scale (5 components) to large-scale (15 components) configurations, and highlights key differences in performance and adaptability under dynamic conditions. Section 7.3.1 presents a comparative analysis with BARON solver (complete problem version) and SPACE4AI-D proposed in [10]. Section 7.3.2 evaluates our full framework against a baseline method for solving the minimum k-cut problem, as well as the approach proposed in [11] which leverages reinforcement learning (PPO), depth-first search, and Dancing Links algorithms.

The parameters used for the scalability analysis are detailed in Table 3. For systems with 5 and 7 components, the action space dimensions are $3^5 = 243$ and $4^7 = 16,384$, respectively, corresponding to configurations with 3 and 4 computational layers. These action space sizes are relatively small, allowing us to utilize PPO without requiring action branching. In contrast, for systems with 10 and 15 components deployed on 5 and 10 computational layers, the action space expands dramatically to 5^{10} and 10^{15}, respectively. These extremely large action spaces render the direct application of PPO infeasible, necessitating the use of PPO with action branching (PPO_AB) to handle the increased complexity effectively. Consequently, in the experiments described in the following subsections, both PPO and PPO_AB are employed for systems with 5 and 7 components. However, for larger-scale systems with 10 and 15 components, only PPO_AB is utilized to ensure computational feasibility and scalability. PPO_AB was trained for 600, 600, 800, and 800 iterations for systems with 5, 7, 10, and 15 components, with average training times of 18, 29, 38, and 106 minutes, respectively. At runtime, the inference time required to solve the joint component placement and resource optimization problem is approximately 0.01 seconds, even for the largest system scale.

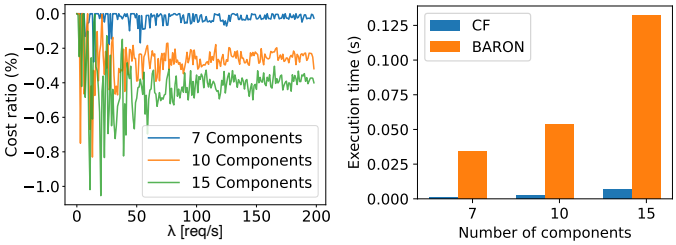
7.3.1 Comparison with BARON and SPACE4AI-D

In this section, we use BARON to address the complete problem defined in (P1), which involves jointly solving the component placement and resource optimization subproblems, treating both x_{ij} and n_j as decision variables. However, the inclusion of x_{ij} as a binary variable significantly increases the computational complexity, rendering BARON impractical for large-scale systems. In contrast, since our proposed approach, with a trained RL model, demonstrates remarkable efficiency, consistently solving the problem within approximately 0.01 seconds, regardless of system scale, the maximum execution time for BARON has been limited up to one hour. As described in [10], SPACE4AI-D employs a random greedy (RG) algorithm to generate placement solutions iteratively. Following this, the algorithm selects the top k-best solutions and refines them



(a) Diff in Opt_n of VMs (CF -BARON) (b) Cost of VMs and cost ratio (c) Response Time (RT) of LCs and GCs by CF (d) Response Time (RT) of LCs and GCs by solver

Figure 5: Comparison of KKT-based closed form (CF) and BARON, single instance of a small system, $\lambda = [0..200]$.



(a) Cost ratio. (b) Execution time.

Figure 6: Comparison of CF and BARON, averaging among 5 instances.

using local search (LS) for a specified number of iterations to further improve their quality. The number of RG iterations must scale with the size of the system due to the exponential growth in the total number of possible placements. To address this, we configured the RG algorithm to perform 5×10^4, 10^5, 2×10^5, and 5×10^5 iterations for scenarios with 5, 7, 10, and 15 components, respectively. From these iterations, the 10-best solutions were selected and subjected to 10^3 iterations of LS for further optimization.

The results, averaged over 5 random instances, are presented in Figure 7. The first row illustrates the cost ratio over time steps for all scenarios, while the second row depicts the varying workloads across time steps and the corresponding execution times required by BARON and SPACE4AI-D to solve the problem. The final row summarizes the average cost ratio across all time steps and the total number of QoS violations incurred by each method. For the 5-component scenario, the cost ratio indicates that PPO and PPO_AB perform slightly worse than BARON and SPACE4AI-D, with average losses of approximately 2% and less than 1%, respectively (see 5 components in Figures 7a and 7c). Despite these differences, all methods exhibit a comparable number of QoS violations. When scaling the system to 7 components, we observed a performance decline in PPO compared to PPO_AB. Specifically, PPO_AB outperformed both BARON and SPACE4AI-D across most workloads, whereas PPO underperformed relative to BARON and SPACE4AI-D in the majority of cases (see the red and green lines vs. the blue and orange lines in Figure 7a). The average cost ratio for PPO_AB across all workloads is approximately 4% with respect to BARON and SPACE4AI-D, demonstrating the lowest violation among all methods, while PPO shows an average cost ratio that is roughly -3% with respect to both (see 7 components in Figure 7c).

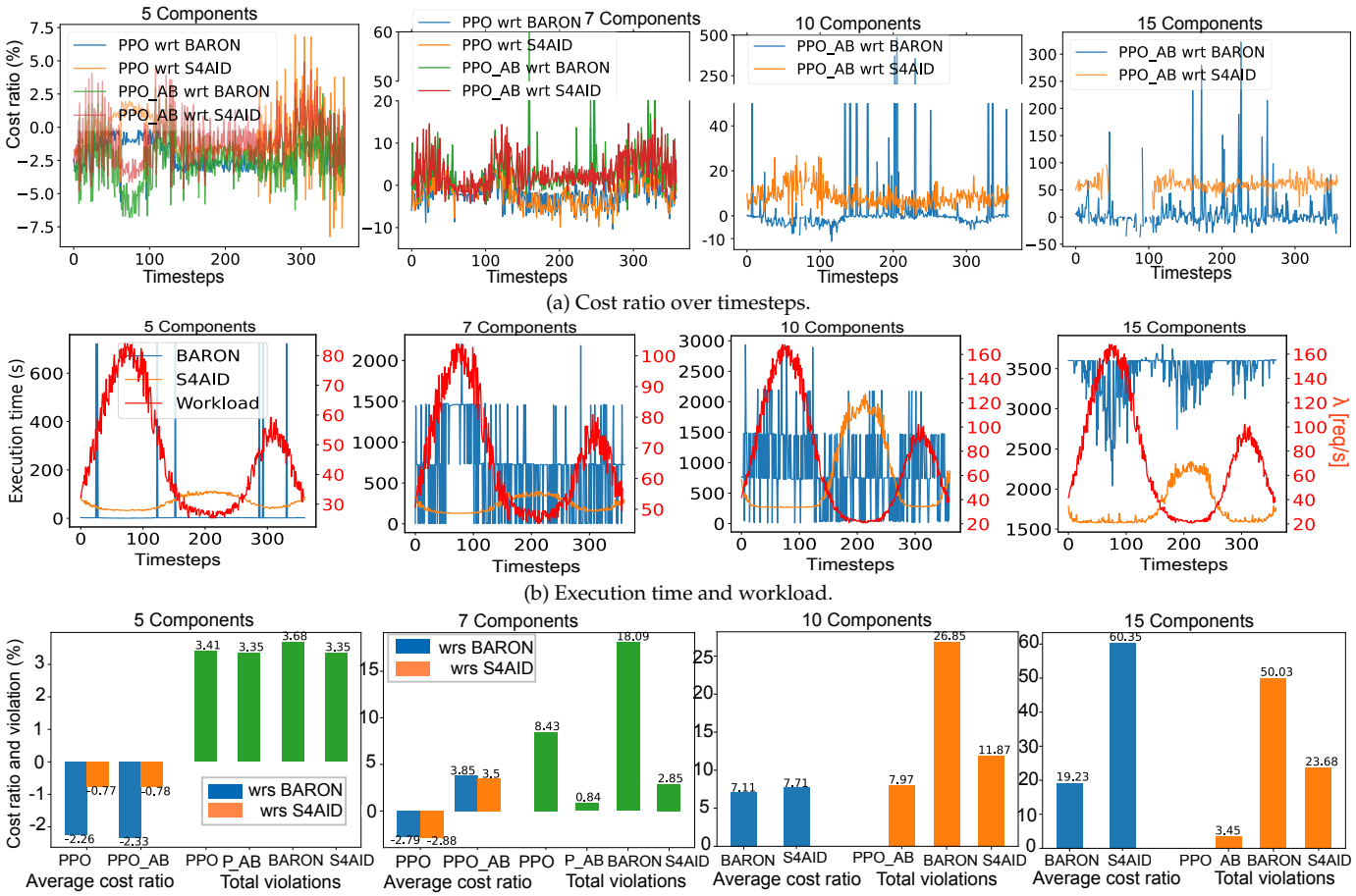
As the system scale increases to 10 and 15 components, the cost ratio of PPO_AB rises significantly across most workloads, with average cost ratios reaching 19% and 60% relative to BARON and SPACE4AI-D, respectively, in the largest-scale scenario (see 10 and 15 components in Figures 7a and 7c). Additionally, a marked difference in total violations is observed between PPO_AB and the other methods, highlighting the capability of PPO_AB to effectively manage larger systems. Note that the disconnected points for 10 and 15 components in Figure 7a indicate scenarios where BARON or SPACE4AI-D failed to find feasible solutions in any of the 5 random instances. This highlights the limitations of SPACE4AI-D in identifying feasible solutions under high workloads. Beyond cost ratio and violations, both BARON and SPACE4AI-D exhibit substantially higher execution times when solving the problem for specific workloads in Figure 7b. For example, SPACE4AI-D requires up to 40 minutes to produce a solution (with approximately 24% violations), whereas BARON frequently exhausts the one-hour time limit, with a 50% violation rate suggesting that one hour was insufficient for half of the cases (see 15 components in Figure 7b).

Based on these findings, PPO_AB demonstrates exceptional scalability and efficiency in managing large systems, solving the problem with significantly lower cost and violations compared to BARON and SPACE4AI-D, all within approximately 0.01 seconds and it is the only method that can be used for the joint component placement and resource optimization at runtime.

7.3.2 *Comparison with PPO_DLX and min-k-cut*

In this section, we further compare our approach with two additional methods: an RL-based approach proposed in [11] and a minimum k-cut-based heuristic method inspired by [41].

Rather than relying on traditional heuristics such as SPACE4AI-D (as discussed in previous subsection), the approach proposed in [11], similar to our method, leverages the dynamics and capabilities of RL, specifically the PPO algorithm, to address the component placement problem for an application DAG. Their work focused on a simplified system without considering QoS performance constraints, assuming homogeneous resource layers and the number of cores as prior requirement for a component. The objective was to group components and assign each group to a resource layer to maximize resource utilization (i.e., the fraction of allocated cores). The exhaustive set of possible groupings was derived by solving the exact cover problem using the Dancing Links technique (DLX) [42]. However, given the exponential growth of possible groupings in large systems, [11] proposed an algorithm to reduce the set of groupings, which employs depth-first search (DFS) on the DAG to eliminate unfeasible groupings that violate the dependencies within the DAG. This reduced set was then utilized as the action space for their RL-based method. The state includes workload translated in the number of users' requests in terms of number of cores to be allocated for an



(a) Cost ratio over timesteps.

(b) Execution time and workload.

(c) Average cost ratio and violation.

Figure 7: Comparison among our full framework, BARON and SPACE4AI-D, average of 5 random instances.

individual component. Since [11] did not consider QoS performance constraints, we adapted our M/G/1 queue-based approach to align with their core-based approach, refer as *PPO_DLX*. To achieve this, as in [11], we assumed homogeneous resources with identical costs across all layers. Each VM instance was modeled as an M/G/1 queue with a single core. We split the global constraint threshold among the components involved in the path proportionally to their respective demand times. In this way, the global constraints were effectively transformed into new local constraints for the components involved in the paths. This reformulation enabled us to determine the optimal number of VM instances using M/G/1 queue system which corresponds to the number of cores in their system. Moreover, we defined a baseline approach based on minimum k-cut [43] along the same lines of the work in [41]. The weights of edges that require cutting correspond to the data size (δ_{ik}) that is transferred between the consecutive components in the DAG (refer to Section 2.1), which are placed in distinct layers. In particular, we utilized the minimum k-cut algorithm to partition the components into k groups, where k corresponds to the total number of layers (as in [41]) according to the scenarios specified in Table 3. Each group was then assigned to a layer. We employed M/G/1 queue system model considering component collocation as defined in our framework. Assuming homogeneous layers, we established both local and global constraints, splitting global constraint threshold using the same method described above for PPO_DLX, and determined the optimal number of VM instances using the method outlined in Section 5.1. It is important to note that, by splitting the global constraints into new local constraints for each component involved in the paths, since we have only local constraints, the problem to obtain minimum number of instances to fulfill the local constraints is convex and we can leverage the closed-form solution presented in Section 5.1 to determine the optimal number of instances in the absence of global constraints. However, since the method used to split the global constraint (proportionally to the components demand times) is greedy, the final solution may deviate from the true global optimum.

The comparison is performed under the scalability analysis scenarios in Table 3. The results, averaged over 5 random instances, are presented in Figure 8. Since this section assumes homogeneous layers, we adjust the workload rate to maintain an average system utilization comparable to that of the scenarios with varying components discussed in the previous section. In the first row, the cost ratio of PPO_AB relative to PPO, PPO_DLX, and the minimum k-cut method is depicted while the second row highlights the average cost ratio and total violations. The total violations in our framework are negligible and consistently lower than those observed with other methods across all scenarios (Figure 8b). The cost ratio across all scenarios is observed to be lower during peak workloads and higher during valleys.

This behavior can be explained by the fact that when the workload is high, both our framework (PPO and PPO_AB) and the min_k_cut approach utilize all available layers to satisfy QoS constraints, as k is fixed and equal to the number of layers (as in [41]). Conversely, during light workloads, our framework optimizes resource usage by employing only a subset of layers, balancing QoS satisfaction with cost minimization. For PPO_DLX, while it dynamically adjusts the total number of groups (and consequently the number of active layers) in response to workload fluctuations, it occasionally relies on component collocation. This oversight leads to an increased number of required VM instances, which in turn raises the cost ratio. The impact is particularly noticeable during periods of light workload, where the total number of required VM instances is inherently smaller, making the cost ratio more pronounced.

As mentioned before, our framework could not execute PPO without action branching for scenarios with 10 and 15 components due to scalability limitations. PPO_DLX faces a similar issue with scaling. For the 10-component scenario, the total number of possible groupings (actions) is $B_{10}^{(5)} = 86472$ [4], which their proposed DFS-based algorithm reduced to approximately 2,200 actions (98% reduction) making it feasible to use PPO in this case. However, for the largest scale scenario with 15 components, the total number of possible groupings exceeds 10^9. Even after significant reduction, the action space remains prohibitively large for standard RL methods, and the action definition in PPO_DLX prevents the application of action branching technique. As a result, PPO_DLX is not scalable, and we compare only the min_k_cut method with PPO_AB for the 15-component scenario (see 15 components in Figures 8a and 8b). The cost ratio trend follows a similar pattern to other scenarios; however, the min_k_cut method exhibits a significant violation of 30%, particularly during peak workloads (disconnected points indicating its inability to find feasible solutions in any of the 5 random instances). This is because the method prioritizes minimizing communication overhead without adequately considering the total response time. In contrast, our framework effectively addresses both objectives and does not experience any violations.

It is worth noting that the relatively lower cost ratio observed compared to Sections 7.3.1 is attributed to the assumption of homogeneous layers, where all VMs have the same cost. This uniformity reduces the impact of component placement decisions, making the agent largely indifferent to specific placement choices. Nevertheless, our framework demonstrated notable improvements in the average cost ratio, outperforming PPO_DLX and the min_k_cut methods by up to 11% and 6%, respectively, in large scale scenarios.

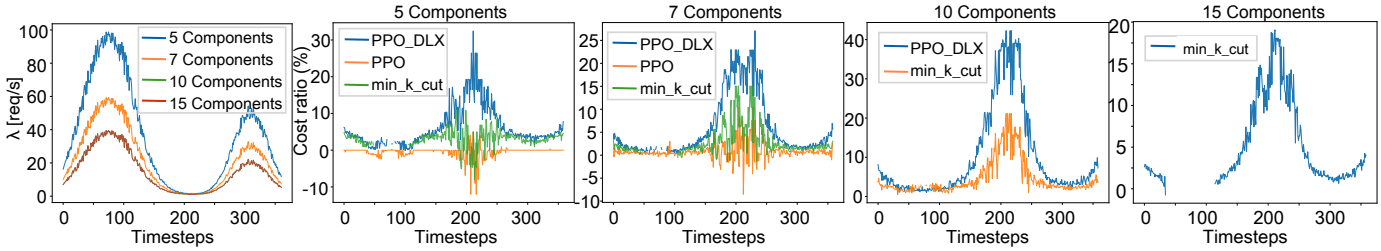
8 Related Work

The problem of component placement and resource optimization in the computing continuum has gained significant attention in recent years due to the rapid proliferation of distributed systems such as cloud, edge, and fog computing. Accordingly, several approaches have been proposed to address these challenges, ranging from heuristic-based methods and optimization frameworks to machine learning-driven techniques [44, 45]. For example, [13] represents task dependencies as DAG and proposed a heuristic-based approach to optimize both task assignment and scheduling based on the priority of tasks. Moreover, efficient resource management for AI applications remains a critical challenge, requiring a balance between dynamic resource demands and high performance in the computing continuum. [30] adopts a Stackelberg game model for real-time resource allocation of AI applications in Mobile Edge Cloud (MEC). This framework models interactions between the MEC platform (leader) and mobile users (followers), enabling a dynamic strategy that adjusts resource provisioning to users' evolving computational demands and AI application requirements. Alternatively, [46] considered hierarchical edge-cloud architectures, where services are modeled as chains of Virtual Network Functions with strict latency and computational requirements. The authors decompose the problem into CPU allocation (solved optimally) and service chain placement (proven NP-hard) subproblems and propose a resource-augmented algorithm that guarantees a feasible solution whenever one exists, while minimizing migration, bandwidth, and computing costs.

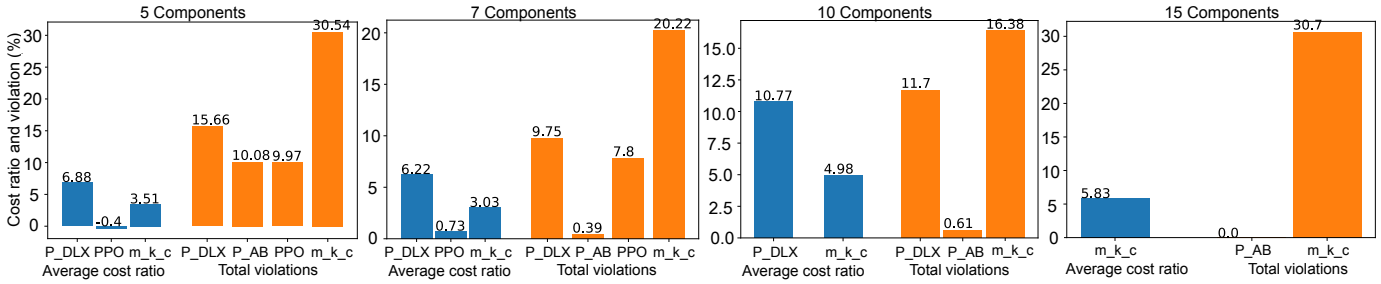
Recent studies have explored the application of RL to address the component placement and resource allocation challenges in the computing continuum. In [23], the authors focus on generating Deep Neural Network (DNN)-based tasks from end devices, considering both lightweight and full DNN models. They formulate an optimization problem aimed at minimizing delay and error while maintaining task queue stability and inference accuracy. A DRL algorithm is employed to make dynamic decisions regarding DNN deployment, task offloading, resource allocation, and channel allocation for each time slot. Additionally, a sub-optimization problem is addressed using Lyapunov optimization to determine the optimal data sizes that meet quality requirements. Differently, [12] proposed a real-time, dependency-aware task offloading approach using Deep Q-Networks (DQN) in the computing continuum, representing mobile applications as DAG and capturing the dependencies and parallelism between tasks. The DQN is tailored to train a dynamic decision-making model for task offloading, enabling rapid generation of offloading strategies in response to changing environments, eliminating the need to preset task priorities, fully accounting for task parallelism during scheduling. Similar to [23] and [12], [14] considers IoT applications modeled as DAG, however, unlike [23] and [12], which use centralized RL methods, [14] proposes an actor-critic-based distributed application placement technique using importance-weighted Actor-Learner architectures to reduce exploration costs. It also features an adaptive off-policy correction mechanism and recurrent layers to capture temporal dynamics. A different approach is taken by [47], which addressed microservice orchestration in a containerized multi-cluster infrastructure. The authors propose a RL environment tailored for Kubernetes clusters, with a multi-objective reward function that optimizes microservice placement by minimizing latency, deployment cost, and placement load imbalance.

To the best of our knowledge, this is the first unified

4. where $B_n^{(l)}$ is Bell number which represents the number of ways to partition a set of n elements into at most l non-empty subsets: $B_n^{(l)} = \sum_{k=0}^{l} s(n,k)$ and $s(n,k)$ is Stirling numbers of the second kind which count the ways to partition n elements into k non-empty subsets.



(a) Cost ratio and workloads over timesteps.



(b) Average cost ratio and violation.

Figure 8: Comparison among our full framework, PPO_DLX and min_k_cut, average of 5 random instances.

Paper	Task dependency	Computational layers	Main focus			Method	
			Objective	QoS guarantee		RL-based	Heuristic
				LC	GC		
[10]	DAG	Heterogeneous	Min resource cost	✓	✓	✗	✓
[11]	DAG	Homogeneous	Max resource utilization	✗	✗	✓	✓
[12]	DAG	Homogeneous	Min RT	✗	✗	✓	✗
[13]	DAG	Homogeneous	Max Num. tasks	✗	✓	✗	✓
[14]	DAG	Heterogeneous	Min Wght. E&RT	✗	✗	✓	✗
[23]	Independent	Homogeneous	Min RT	✗	✗	✓	✗
[30]	Independent	Homogeneous	Min resource cost	✓	✗	✗	✓
[46]	Chain	Heterogeneous	Min resource cost	✗	✓	✗	✓
[47]	Independent	Heterogeneous	Min Wght. cost&RT	✗	✗	✗	✗
Our approach	DAG	Heterogeneous	Min resource cost	✓	✓	✓	✓

Table 2: Positioning of our approach with relevant works. Wght: weighted, Num: Number, RT: response time, E: energy, LC: local constraint, GC: global constraint.

optimization framework that combines analytical models accounting for resource contention in application performance estimation, based on queuing theory, with a hybrid optimization technique incorporating KKT conditions and RL. Furthermore, our research addresses a research gap in the existing literature by applying RL to the placement problem, utilizing a branching dueling architecture that reduces the exponential action space to a linear one, thus improving the efficiency and scalability of the solution. The detailed comparison of related work and our proposed method is shown in Table 2.

9 Conclusions

In this paper, we modeled the component placement and resource optimization problem in computing continua as an MINLP. We decomposed the problem into two subproblems, obtaining the optimal number of VM instances through KKT conditions and the optimal placement through reinforcement learning. Extensive experimental analyses demonstrated that the proposed framework achieved significant cost reductions, averaging 19% and 60%, compared to BARON and SPACE4AI-D, respectively, in the largest-scale scenarios. Future work includes integrating the resource optimization solution with a simulator to enable RL-based dynamic adaptation to real-world component service demand distributions. Additionally, we aim to extend the framework to address AI-driven applications where each component represents a partitionable DNN with multiple alternative deployment options. These alternatives would enable dynamic selection based on workload fluctuations at runtime, ensuring adaptability, resource efficiency, and QoS compliance. We also plan to integrate our framework with mainstream orchestration platforms such as Kubernetes, enabling practical deployment and evaluation in real-world cloud–edge environments.

References

[1] T. H. Luan et al., "Fog computing: Focusing on mobile users at the edge," *IEEE Communications Magazine*, vol. 54, no. 8, 2018.
[2] D. Schubmehl, *Worldwide Artificial Intelligence Software Platforms Forecast, 2023–2027*, https : / / www . idc . com / getdoc . jsp ? containerId=US51044723, 2023.
[3] B. Varghese et al., "Challenges and opportunities in edge computing," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 30–36, 2016.
[4] H. Sedghani et al., "Advancing design and runtime management of ai applications with ai-sprint (position paper)," in *2021 IEEE 45th COMPSAC*, 2021, pp. 1455–1462.
[5] H. Sedghani et al., "A randomized greedy method for ai applications component placement and resource selection in computing continua," in *IEEE JCC*, 2021, pp. 65–70.
[6] X. Qin et al., "Erth scheduler: Enhanced red-tailed hawk algorithm for multi-cost optimization in cloud task scheduling," *Artificial Intelligence Review*, vol. 57, no. 328, 2024.
[7] F. Filippini et al., "Space4ai-r: A runtime management tool for ai applications component placement and resource scaling in computing continua," in *IEEE/ACM 16th UCC*, 2024.

[8] B. P. Rimal et al., "Ai-orchestrated edge–cloud continuum for autonomous management of smart environments," *IEEE Internet of Things Journal*, vol. 10, no. 4, pp. 2708–2724, 2023.
[9] *MINLP:BARON*, https://minlp.com/baron-solver, 2022.
[10] H. Sedghani et al., "Space4ai-d: A design-time tool for ai applications resource selection in computing continua," *IEEE Transactions on Services Computing*, vol. 17, no. 6, pp. 4324–4339, 2024.
[11] E. Paraskevoulakou et al., "Enhancing cloud-based application component placement with ai-driven operations," in *2024 IEEE 14th CCWC*, 2024, pp. 0687–0694.
[12] X. Chen et al., "Real-time offloading for dependent and parallel tasks in cloud-edge environments using deep reinforcement learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 3, pp. 391–404, 2024.
[13] H. Liao et al., "Dependency-aware application assigning and scheduling in edge computing," *IEEE Internet of Things Journal*, vol. 9, no. 6, pp. 4451–4463, 2022.
[14] M. Goudarzi et al., "A distributed deep reinforcement learning technique for application placement in edge and fog computing environments," *IEEE Transactions on Mobile Computing*, vol. 22, no. 5, pp. 2491–2505, 2023.
[15] D. Ardagna et al., "Adaptive service composition in flexible processes," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 369–384, 2007.
[16] T. Elgamal et al., "Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement," in *IEEE/ACM(SEC))*, 2018.
[17] L. Bao et al., "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, 2019.
[18] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX ATC 20*, USENIX Association, 2020, pp. 205–218.
[19] U. Tadakamalla et al., "Autonomic resource management for fog computing," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2021.
[20] *Amazon EC2 On-Demand Pricing*, https://aws.amazon.com/ec2/pricing/on-demand/.
[21] *Pricing calculator*, https://azure.microsoft.com/en-us/pricing/calculator/.
[22] C. Zaw et al., "Radio and computing resource allocation in co-located edge computing: A generalized nash equilibrium model," *IEEE Transactions on Mobile Computing*, vol. 22, no. 4, pp. 2340–2352, 2023.
[23] W. Fan et al., "Dnn deployment, task offloading, and resource allocation for joint task inference in iiot," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 2, pp. 1634–1646, 2023.
[24] C. Chen et al., "Latency minimization for mobile edge computing networks," *IEEE Transactions on Mobile Computing*, vol. 22, no. 4, pp. 2233–2247, 2023.
[25] E. D. Lazowska et al., *Quantitative system performance: computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.
[26] M. Kandpal et al., "Role of predictive modeling in cloud services pricing: A survey," in *IEEE Confluence*, 2017.
[27] B. E. Zant et al., "Performance and price analysis for cloud service providers," in *IEEE (SAI)*, 2015.
[28] *AWS Products*, https://aws.amazon.com/products/.
[29] *Azure Products*, https://azure.microsoft.com/.
[30] R. Sala et al., "AI Applications Resource Allocation in Computing Continuum: A Stackelberg Game Approach," *IEEE Transactions on Cloud Computing*, vol. 13, no. 1, pp. 166–183, 2025.
[31] H. Sedghani et al., "A Stackelberg game approach for managing AI sensing tasks in mobile crowdsensing," *IEEE Access*, vol. 10, pp. 91 524–91 544, 2022.
[32] J. S. Arora et al., "Hierarchical decomposition of optimization problems," *Journal of Optimization Theory and Applications*, vol. 45, no. 2, pp. 173–187, 1985.
[33] A. Tavakoli et al., "Action branching architectures for deep reinforcement learning," ser. AAAI'18, New Orleans, Louisiana, USA: AAAI Press, 2018.
[34] Z. Wang et al., "Dueling network architectures for deep reinforcement learning," ser. ICML'16, JMLR.org, 2016, pp. 1995–2003.
[35] E. Liang et al., "RLlib: Abstractions for distributed reinforcement learning," in *ICML*, 2018.
[36] J. Schulman et al., "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.
[37] P. Mair et al., "Robust statistical methods in R using the WRS2 package," *Behavior Research Methods*, vol. 52, pp. 464–488, 2020.
[38] C. Lin et al., "Modeling and optimization of performance and cost of serverless applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 615–632, 2021.
[39] A. Kambale et al., "Runtime management of artificial intelligence applications for smart eyewears," in *IEEE/ACM UCC*, 2024.
[40] Y. Chen et al., "Analysis and lessons from a publicly available google cluster trace," Berkeley, Tech. Rep. UCB/EECS-2010-95, 2010.
[41] K. Fu et al., "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2022.
[42] D. E. Knuth, *Dancing links*, 2000. arXiv: cs / 0011047. [Online]. Available: https://arxiv.org/abs/cs/0011047.
[43] D. R. Karger et al., "A new approach to the minimum cut problem," *Journal of the ACM*, vol. 43, no. 4, pp. 601–640, 1996.
[44] S. Smolka et al., "Evaluation of fog application placement algorithms: A survey," *Computing*, vol. 104, no. 6, pp. 1397–1423, 2022.
[45] M. Ghorbian et al., "Function placement approaches in serverless computing: A survey," *Journal of Systems Architecture*, vol. 157, p. 103 291, 2024.
[46] I. Cohen et al., "Dynamic service provisioning in the edge-cloud continuum with bounded resources," *IEEE/ACM Transactions on Networking*, vol. 31, no. 6, pp. 3096–3111, 2023.
[47] J. Santos et al., "Hephaestusforge: Optimal microservice deployment across the compute continuum via reinforcement learning," *Future Generation Computer Systems*, vol. 166, p. 107 680, 2025.

Hamta Sedghani received her B.Sc. degree from Iran University of Science and Technology, Tehran, Iran and her M.S and Ph.D. degrees from University of Tabriz, Tabriz, Iran in Computer Engineering. Currently, she is post-doc researcher at Politecnico di Milano. Her current interests include game theory, optimization and reinforcement learning for resource management in computing continuum.

Mauro Passacantando received the M.S. and Ph.D. degrees in Mathematics from University of Pisa, Italy. He is currently an Associate Professor of Operations Research (qualified for Full Professorship) with the Department of Business and Law, University of Milano-Bicocca. His research is mainly devoted to variational inequalities and equilibrium problems. In recent years, his work has also focused on warehouse management and green logistics problems. He has published more than 80 peer-reviewed papers in international journals, book chapters and conference proceedings.

Danilo Ardagna is Associate Professor at the Dipartimento di Elettronica Informazione and Bioingegneria at Politecnico di Milano. He received a Ph.D. degree in computer engineering in 2004 from Politecnico di Milano from which he also graduated in December 2000. His work focuses on the design, prototype and evaluation of optimization algorithms for resource management of cloud computing and big data systems.

APPENDIX

In the following subsections, we provide the proofs of the theorems and the tables of the experimental setup and ANCOVA test.

Proofs of the theorems

We provide the proofs of the theorems stated in Sections 3, 4 and 5, and we outline the idea behind the correctness of Algorithm 1. We recall that Theorems 3.1 and 4.1 state the non-convexity of constraints (P1g) in problem (P1) and convexity of problem (P2), respectively. Theorem 5.1 investigates all possible conditions related to the global constraint P and if the problem is feasible, it provides an optimal number of additional instances across all layers for a specific resource partitioning.

Proof of Theorem 3.1

Proof. Recall that the global constraint (P1g) has a term that contains the product of two binary decision variables:

$$f(x) = \sum_{\substack{i,k\in P:\\ i\neq k}} \frac{\delta_{ik}}{g(x)}, \quad \text{where} \quad g(x) = \sum_{\substack{j,l\in\mathcal{J},\\ j\neq l}} B_{jl}x_{ij}x_{kl}.$$

The first order partial derivatives of f with respect to the variables x_{ij} are

$$\frac{\partial f}{\partial x_{ij}} = \sum_{\substack{i,k\in P:\\ i\neq k}} -\delta_{ik}\frac{1}{g(x)^2}\frac{\partial g(x)}{\partial x_{ij}},$$

where

$$\frac{\partial g(x)}{\partial x_{ij}} = \sum_{\substack{l\in\mathcal{J},\\ l\neq j}} B_{jl}x_{kl},$$

thus

$$\frac{\partial f}{\partial x_{ij}} = \sum_{\substack{i,k\in P:\\ i\neq k}} -\delta_{ik}\frac{1}{g(x)^2}\sum_{\substack{l\in\mathcal{J},\\ l\neq j}} B_{jl}x_{kl}$$

The second order partial derivatives of f are

$$\frac{\partial^2 f}{\partial x_{ij}\partial x_{kl}} = \frac{\partial}{\partial x_{kl}}\left(\sum_{\substack{i,k\in P:\\ i\neq k}} -\delta_{ik}\frac{1}{g(x)^2}\sum_{\substack{l\in\mathcal{J},\\ l\neq j}} B_{jl}x_{kl}\right).$$

The first term $\frac{1}{g(x)^2}$ involves $g(x)$, which is the sum of bilinear terms; hence its partial derivative with respect to x_{kl} is

$$\frac{\partial}{\partial x_{kl}}\left(\frac{1}{g(x)^2}\right) = -2\frac{1}{g(x)^3}\frac{\partial g(x)}{\partial x_{kl}}.$$

The second term $\sum_{\substack{l\in\mathcal{J},\\ l\neq j}} B_{jl}x_{kl}$ is linear, so its partial derivative with respect to x_{kl} is

$$\frac{\partial}{\partial x_{kl}}\left(\sum_{\substack{l\in\mathcal{J},\\ l\neq j}} B_{jl}x_{kl}\right) = B_{jl}.$$

Therefore, we get

$$\frac{\partial^2 f}{\partial x_{ij}\partial x_{kl}} = \sum_{\substack{i,k\in P:\\ i\neq k}} \delta_{ik}\left[\frac{2}{g(x)^3}\sum_{\substack{l\in\mathcal{J},\\ l\neq j}} B_{jl}x_{kl}\sum_{\substack{m\in\mathcal{J},\\ m\neq k}} B_{jm}x_{il} - \frac{B_{jl}}{g(x)^2}\right].$$

The Hessian matrix above is indefinite due to the presence of mixed second-order partial derivatives involving products of the decision variables $x_{ij}x_{kl}$. Hence, the global constraint (P1g) is non-convex. □

Proof of Theorem 4.1

Proof. To prove that problem (P2) is convex, it is sufficient to prove that the function

$$f(n_j) = \frac{n_j}{n_j - L_j}$$

is convex when $n_j > L_j$. The first derivative of f is

$$f'(n_j) = -\frac{L_j}{(n_j - L_j)^2},$$

while the second derivative of f is

$$f''(n_j) = \frac{2L_j}{(n_j - L_j)^3} > 0,$$

thus f is convex. □

Proof of Theorem 5.1

Proof. We remark that

$$\sum_{j\in\mathcal{J}_P}\frac{(NC_j + n'_j)Z_{Pj}}{NC_j + n'_j - L_j} \geq \sum_{j\in\mathcal{J}_P}\frac{(NC_j + N'_j)Z_{Pj}}{NC_j + N'_j - L_j}$$

holds for any $n'_j \in [0, N'_j]$. Hence, if $\sum_{j\in\mathcal{J}_P}\frac{(NC_j+N'_j)Z_{Pj}}{NC_j+N'_j-L_j} > TH_P$ holds, then constraint (P3b) cannot be satisfied and problem (P3) is infeasible.

Problem (P3) is convex because its objective function is linear and constraint (P3b) is convex. Moreover, it admits an optimal solution and the Slater constraint qualification holds. Hence, problem (P3) is equivalent to solve the corresponding KKT system:

$$\begin{aligned}
&c_j - \mu\frac{L_j Z_{Pj}}{(NC_j + n'_j - L_j)^2} - \nu_j + \delta_j = 0 && \forall\, j\in\mathcal{J}_P\\
&\mu\left[\sum_{j\in\mathcal{J}}\frac{(NC_j + n'_j)Z_{Pj}}{NC_j + n'_j - L_j} - TH_P\right] = 0 &&\\
&\nu_j n'_j = 0 && \forall\, j\in\mathcal{J}_P\\
&\delta_j(N'_j - n'_j) = 0 && \forall\, j\in\mathcal{J}_P\\
&\sum_{j\in\mathcal{J}}\frac{(NC_j + n'_j)Z_{Pj}}{NC_j + n'_j - L_j} \leq TH_P &&\\
&0 \leq n'_j \leq N'_j && \forall\, j\in\mathcal{J}_P\\
&\mu \geq 0,\ \nu_j \geq 0,\ \delta_j \geq 0 && \forall\, j\in\mathcal{J}_P
\end{aligned}$$

If

$$\sum_{j\in\mathcal{J}_P}\frac{NC_j\, Z_{Pj}}{NC_j - L_j} \leq TH_P,$$

then the optimal solution is $n'^*_j = 0$ for all $j\in\mathcal{J}_P$, with $\mu = \delta_j = 0$ and $\nu_j = c_j$ for any $j\in\mathcal{J}_P$.

If

$$\sum_{j\in\mathcal{J}_P}\frac{NC_j\, Z_{Pj}}{NC_j - L_j} > TH_P,$$

then $\mu > 0$. We define the following subsets of indices related to the optimal solution n'^*:

$$\begin{aligned}
\mathcal{J}_P^0 &:= \{j\in\mathcal{J}_P :\ n'^*_j = 0\},\\
\mathcal{J}_P^{>} &:= \{j\in\mathcal{J}_P :\ 0 < n'^*_j < N'_j\},\\
\mathcal{J}_P^{=} &:= \{j\in\mathcal{J}_P :\ n'^*_j = N'_j\}.
\end{aligned}$$

For any $j\in\mathcal{J}_P^0$, one has $n'^*_j = 0$, $\delta_j = 0$ and $\nu_j \geq 0$, hence

$$\sqrt{\mu} \leq \frac{NC_j - L_j}{\sqrt{L_j Z_{Pj}/c_j}}. \tag{29}$$

For any $j\in\mathcal{J}_P^{>}$ we have $\nu_j = \delta_j = 0$, hence

$$n'^*_j = L_j - NC_j + \sqrt{\mu}\sqrt{L_j Z_{Pj}/c_j}. \tag{30}$$

Scenario	#Components	#Computational layers	Bimodal trace range λ (req/s)	#Local and global constraints	Activation function	Neurons per layer	#Training iterations
1	5	Edge: 1, Cloud: 2	[25, 85]	3, 3	[ReLU, ReLU]	[256, 128]	600
2	7	Edge: 2, Cloud: 2	[45, 105]	4, 4	[ReLU, ReLU]	[512, 256]	600
3	10	Edge: 2, Cloud: 3	[20, 170]	5, 5	[ReLU, ReLU, ReLU]	[512, 256, 128]	800
4	15	Edge: 5, Cloud: 5	[20, 170]	7, 7	[ReLU, ReLU, ReLU]	[1024, 512, 256]	800

Table 3: Parameters of scalability analysis.

Moreover, $0 < n'^*_j < N'_j$ implies

$$\frac{NC_j - L_j}{\sqrt{L_j Z_{Pj}/c_j}} < \sqrt{\mu} < \frac{NC_j + N'_j - L_j}{\sqrt{L_j Z_{Pj}/c_j}}. \tag{31}$$

For any $j \in \mathcal{J}_P^=$, one has $n'^*_j = N'_j$, $\nu_j = 0$ and $\delta_j \geq 0$, hence

$$\sqrt{\mu} \geq \frac{NC_j + N'_j - L_j}{\sqrt{L_j Z_{Pj}/c_j}}. \tag{32}$$

Since $\mu > 0$, we get

$$\begin{aligned} TH_P &= \sum_{j \in \mathcal{J}_P} \frac{(NC_j + n'_j) Z_{Pj}}{NC_j + n'_j - L_j} \\ &= \sum_{j \in \mathcal{J}_P^>} \frac{L_j + \sqrt{\mu}\sqrt{L_j Z_{Pj}/c_j}}{\sqrt{\mu}\sqrt{L_j Z_{Pj}/c_j}} Z_{Pj} + \sum_{j \in \mathcal{J}_P^0} \frac{NC_j Z_{Pj}}{NC_j - L_j} \\ &\quad + \sum_{j \in \mathcal{J}_P^=} \frac{(NC_j + N'_j) Z_{Pj}}{NC_j + N'_j - L_j} \\ &= \sum_{j \in \mathcal{J}_P^>} Z_{Pj} + \frac{1}{\sqrt{\mu}} \sum_{j \in \mathcal{J}_P^>} \sqrt{L_j Z_{Pj} c_j} + \sum_{j \in \mathcal{J}_P^0} \frac{NC_j Z_{Pj}}{NC_j - L_j} \\ &\quad + \sum_{j \in \mathcal{J}_P^=} \frac{(NC_j + N'_j) Z_{Pj}}{NC_j + N'_j - L_j} \end{aligned}$$

Hence,

$$\sqrt{\mu} = \frac{\sum_{j \in \mathcal{J}_P^>} \sqrt{L_j Z_{Pj} c_j}}{TH_P - \sum_{j \in \mathcal{J}_P^>} Z_{Pj} - \sum_{j \in \mathcal{J}_P^0} \frac{NC_j Z_{Pj}}{NC_j - L_j} - \sum_{j \in \mathcal{J}_P^=} \frac{(NC_j + N'_j) Z_{Pj}}{NC_j + N'_j - L_j}} := \alpha.$$

The definition of the sets $\mathcal{J}_P^0$, $\mathcal{J}_P^>$, $\mathcal{J}_P^=$ and (30) imply that the optimal solution n'^* is given by formula (18). Finally, (29), (31) and (32) guarantee that α satisfies conditions (20)–(22). □

Optimality of Algorithm 1

The conditions (20)–(22) in Theorem 5.1 guarantee that the indices corresponding to the highest values of β_j belong to the set $\mathcal{J}_P^0$, the indices corresponding to the lowest values of γ_j belong to the set $\mathcal{J}_P^=$, while the remaining indices belong to $\mathcal{J}_P^>$. For this reason, in Algorithm 1 we set $\mathcal{J}_P^=$ equal to the set of indices corresponding to the h smallest values of γ_j (with $h \in \{0, \ldots, |\mathcal{J}_P|\}$), $\mathcal{J}_P^0$ equal to the set of indices corresponding to the k highest values of β_j (with $k \in \{0, \ldots, |\mathcal{J}_P|\}$), and $\mathcal{J}_P^>$ equal to the set of remaining indices. Since the optimal partition exists, the double for loop in Algorithm 1 allows finding it after at most $O(|\mathcal{J}_P|^2)$ iterations. Figure 9 shows an example of the optimal partition $\{\mathcal{J}_P^0, \mathcal{J}_P^>, \mathcal{J}_P^=\}$ based on the position of α with respect to the values of β_j and γ_j.

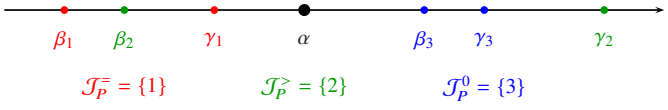


Figure 9: Example of optimal partition $\{\mathcal{J}_P^0, \mathcal{J}_P^>, \mathcal{J}_P^=\}$.

System Parameters	
$\lvert\mathcal{I}\rvert$	{5, 7, 10, 15}
D_{ij}	Uniform distribution in [0.1, 0.5] s
c_j	Uniform distribution in [0.5, 1] $/h
N_j	Uniform distribution in [50, 100]
Agent Parameters	
Episode length	360 steps
lr	decaying from 0.01 to 0.001 during 3000000 steps
γ	0.1
β	0.7
$train\_batch\_size$	4096
$sgd\_minibatch\_size$	512

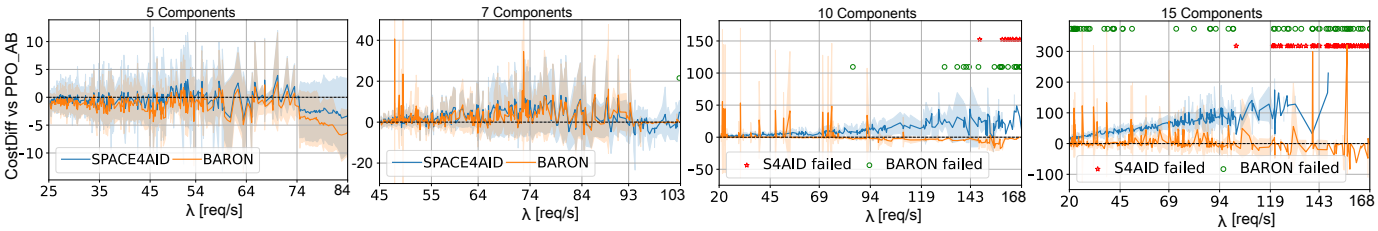
Table 4: Simulation parameters

Experimental setup parameters

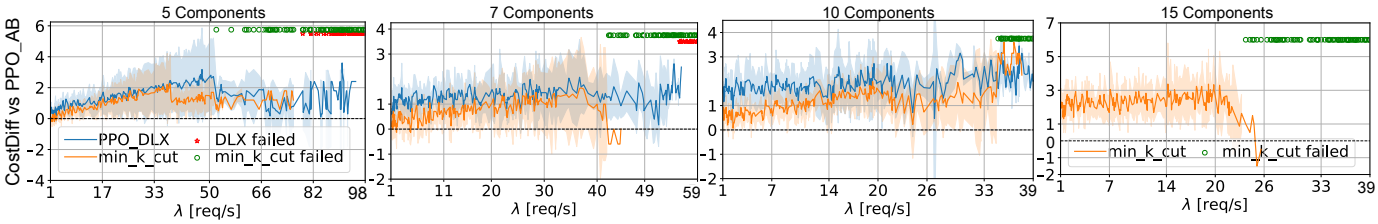
Table 3 presents the detailed system in the scalability analysis and Table 4 outlines the key parameters used in our numerical analyses.

Statistical Analysis

To evaluate the effectiveness of PPO_AB in minimizing operational costs, we conducted a statistical comparison against four baseline methods: BARON, SPACE4AI-D, PPO_DLX and min_k_cut. The analysis was performed across four representative scenarios characterized by increasing system complexity (5, 7, 10, and 15 components and five random instances for each scenario as mentioned in Section 7.1). Figure 10a illustrates the cost difference between PPO_AB and the two baselines, BARON, SPACE4AI-D, across workloads for each scenario. Each line represents the mean cost difference over five instances, with the shaded region indicating the standard deviation. Positive values indicate that PPO_AB outperformed the respective method. Notably, the shaded regions are narrower in the more complex scenarios (10 and 15 components), particularly at higher workload levels. This reduction in shading does not indicate lower variability but is instead due to a decrease in the number of available data points. Specifically, instances where a baseline method failed to produce a feasible solution were excluded from the analysis. For example, in the 15-component scenario, SPACE4AI-D was unable to find feasible solutions for any instances with workloads above 147, resulting in an early termination of its cost curve. Markers in the plots denote workload levels at which a method failed to find feasible solutions in at least four out of five instances. Figure 10b illustrates the cost difference between PPO_AB and two additional baseline methods, PPO_DLX and min_k_cut, across varying workload levels for each scenario. As system complexity increases, both PPO_DLX and min_k_cut exhibit a growing number of infeasible cases, especially under high workloads. These frequent infeasibilities, particularly visible for min_k_cut in the 15-component scenario (where PPO_DLX is omitted due to its prohibitively large action space, which renders it inapplicable), underscore the limitations of these baselines in maintaining QoS constraints, whereas PPO_AB consistently produces feasible and cost-effective solutions.



(a) BARON and SPACE4AI-D.



(b) PPO_DLX and min_k_cut.

Figure 10: Cost difference ($OtherMethod - PPO\_AB$) across workloads. Lines show the mean over 5 instances, shaded areas indicate standard deviation. Markers highlight workloads where the methods failed to find a feasible solution in at least 4 out of 5 instances.

To quantify the statistical significance of these differences, we performed a *robust ANCOVA* (Analysis of Covariance) test [37], treating workload as a covariate. Robust ANCOVA estimates the difference in average cost between methods across varying workload thresholds, reporting both confidence intervals (CIs) and corresponding p-values. A CI, such as $[a, b]$ with a p-value of 0.01, implies that there is a 99% level of confidence that the true cost difference lies within that range. Crucially, when the CI excludes zero—particularly at conventional significance levels (e.g., $p < 0.05$ or $p < 0.01$)—this indicates a statistically significant difference in performance between the methods being compared.

The results are summarized in Table 5 and Table 6, which report the estimated cost difference between PPO_AB and each baseline at five systematically chosen representative workload levels. These points are selected internally by the WRS2 package to summarize the covariate (in this case, Workload) and facilitate interpretation. The tables also report the corresponding confidence intervals and p-values. In Table 5, the ANCOVA results confirm that PPO_AB consistently achieves lower or competitive costs compared to BARON and SPACE4AI-D. For the most complex scenario (15 components), the differences are not only larger but also statistically significant across all workload levels ($p\_value < 0.001$). In simpler scenarios (7 and 10 components), PPO_AB performs comparably to SPACE4AI-D, with differences becoming significant in the higher workload ranges. Against BARON, PPO_AB outperforms even in low-complexity settings (7 and 10 components). These results demonstrate the robustness and scalability of PPO_AB, particularly in scenarios where traditional optimization methods struggle to find feasible or cost-effective solutions.

Table 6 presents a detailed statistical analysis of the cost differences between PPO_AB and the two baselines, PPO_DLX and min_k_cut. Although PPO_AB consistently achieves lower costs than PPO_DLX and min_k_cut across all scenarios (as indicated by the positive Diff values), the differences are not always statistically significant according to the ANCOVA test. This can be attributed to two factors. First, the assumption of homogeneous resources with identical costs across all layers—necessitated by the inability of the alternative methods to handle heterogeneous environments (see Section 7.3.2)—diminishes the influence of component placement decisions, thereby narrowing the cost differentials between methods. Second and most importantly, ANCOVA only considers feasible solutions — that is, solutions that meet the QoS constraints. In contrast, the PPO_DLX and min_k_cut methods often incur substantial QoS violations, particularly in large-scale or high-workload scenarios. These infeasible runs are excluded from the ANCOVA analysis, meaning the comparison is based only on their best-performing (feasible) instances. Consequently, the ANCOVA test underestimates the practical advantage of PPO_AB, which consistently yields feasible solutions with significantly no violations. In real deployment scenarios, where feasibility is essential, PPO_AB demonstrates clear superiority despite conservative statistical significance in some test cases.

Scenario	$\lambda[req/s]$	vs BARON			vs SPACE4AI-D		
		Diff	CI	p-value	Diff	CI	p-value
5 Comps	25.28	-1.0449	[-2.2522, 0.1624]	0.0263	-0.5893	[-1.8452, 0.6667]	0.2281
	30.21	-1.0798	[-2.5185, 0.3588]	0.0539	-0.5657	[-2.0326, 0.9012]	0.3218
	43.42	-1.0092	[-2.8078, 0.7893]	0.1494	-0.3102	[-2.1477, 1.5272]	0.6644
	57.48	-0.9090	[-3.3127, 1.4947]	0.3313	0.1304	[-2.3640, 2.6248]	0.8931
	84.08	-2.4320	[-7.5582, 2.6941]	0.2234	-0.8288	[-6.0628, 4.4052]	0.6842
7 Comps	45.31	4.3040	[2.9290, 5.6789]	0.0000	1.6518	[0.5000, 2.8036]	0.0002
	51.35	5.0379	[3.3083, 6.7675]	0.0000	1.5887	[0.1554, 3.0219]	0.0045
	63.53	6.4227	[4.2770, 8.5684]	0.0000	2.1264	[0.2715, 3.9812]	0.0033
	78.10	9.9806	[6.5130,13.4482]	0.0000	4.6482	[1.9023, 7.3941]	0.0000
	103.94	43.7517	[33.3007,54.2026]	0.0000	5.5943	[0.2534,10.9353]	0.0075
10 Comps	20.64	8.9085	[4.3870, 13.4299]	0.0000	2.9190	[-0.9622, 6.8002]	0.0535
	31.84	11.8843	[6.2591, 17.5095]	0.0000	3.4593	[-1.1473, 8.0659]	0.0538
	65.81	16.4093	[8.8757, 23.9429]	0.0000	4.3530	[-1.6157,10.3216]	0.0611
	100.55	43.3407	[29.3800,57.3014]	0.0000	9.7334	[2.3599,17.1070]	0.0007
	168.02	35.0044	[18.8584,51.1505]	0.0000	31.5321	[16.2192,46.8449]	0.0000
15 Comps	20.64	263.1518	[227.5357,298.7680]	0.0000	31.3049	[26.5777,36.0322]	0.0000
	31.84	255.0906	[222.9551,287.2261]	0.0000	36.0979	[30.2831,41.9128]	0.0000
	65.81	238.3058	[210.8436,265.7679]	0.0000	45.7484	[38.3317,53.1650]	0.0000
	100.55	207.6311	[176.1996,239.0627]	0.0000	107.8733	[89.4390,126.3075]	0.0000
	168.02	288.4801	[254.5853,322.3748]	0.0000	240.6555	[235.3294,245.9815]	0.0000

Table 5: ANCOVA results comparing `PPO_AB` vs `BARON` and `SPACE4AI-D` across scenarios. Differences represent the cost of comparator minus the cost of PPO_AB.

Scenario	λ [req/s]	vs PPO_DLX			vs min_k_cut		
		Diff	CI	p-value	Diff	CI	p-value
5 Comps	1.27	0.8537	[-1.2796, 2.9870]	0.3040	0.5033	[-1.6490, 2.6556]	0.5480
	9.02	1.0066	[-1.5890, 3.6023]	0.3191	0.6590	[-1.9307, 3.2487]	0.5133
	31.17	1.2395	[-1.9654, 4.4444]	0.3204	0.9533	[-2.2780, 4.1846]	0.4484
	53.60	1.9389	[-1.6091, 5.4870]	0.1605	16.2284	[10.6286, 21.8283]	0.0000
	98.83	6.9412	[3.3377, 10.5447]	0.0000	9.9421	[7.1890, 12.6952]	0.0000
7 Comps	1.12	1.1827	[-1.4814, 3.8468]	0.2542	0.4523	[-2.2197, 3.1243]	0.6636
	5.82	1.2136	[-2.0682, 4.4954]	0.3421	0.5379	[-2.7578, 3.8336]	0.6750
	18.98	1.2456	[-2.8316, 5.3229]	0.4325	0.6664	[-3.4205, 4.7533]	0.6752
	32.35	1.4139	[-2.6859, 5.5137]	0.3757	1.6147	[-2.9485, 6.1778]	0.3634
	59.30	17.2312	[10.6671, 23.7954]	0.0000	42.7992	[39.8231, 45.7753]	0.0000
10 Comps	1.05	1.7948	[-0.4168, 4.0065]	0.0372	0.7322	[-1.4751, 2.9394]	0.3941
	4.22	1.8080	[-0.8424, 4.4584]	0.0798	0.7938	[-1.8653, 3.4530]	0.4431
	12.89	1.8666	[-1.4527, 5.1859]	0.1486	0.9414	[-2.3817, 4.2645]	0.4667
	21.73	2.7089	[-1.3037, 6.7215]	0.0830	1.3716	[-2.4159, 5.1591]	0.3522
	39.53	43.6529	[35.0983, 52.2075]	0.0000	59.6729	[50.1918, 69.1540]	0.0000
15 Comps	1.05	—	—	—	2.2154	[-4.0022, 8.4330]	0.3600
	4.22	—	—	—	2.2592	[-5.4602, 9.9785]	0.4521
	12.89	—	—	—	2.7635	[-7.0413,12.5684]	0.4689
	21.73	—	—	—	101.2875	[75.1686,127.4065]	0.0000
	39.53	—	—	—	176.6480	[165.8421,187.4539]	0.0000

Table 6: ANCOVA results comparing `PPO_AB` vs `PPO_DLX` and `min_k_cut` across scenarios. Differences represent the cost of comparator minus the cost of PPO_AB.