SPACE4AI-D: A Design-time Tool for AI Applications Resource Selection in Computing Continua

Hamta Sedghani, Federica Filippini, Danilo Ardagna

Abstract—Nowadays, Artificial Intelligence (AI) applications are becoming increasingly popular in a wide range of industries, mainly thanks to Deep Neural Networks (DNNs) that needs powerful resources. Cloud computing is a promising approach to serve AI applications thanks to its high processing power, but this sometimes results in an unacceptable latency because of long-distance communication. Vice versa, edge computing is close to where data are generated and therefore it is becoming crucial for their timely, flexible, and secure management. Given the more distributed nature of the edge and the heterogeneity of its resources, efficient component placement and resource allocation approaches become critical in orchestrating the application execution.
In this paper, we formulate the resource selection and AI applications component placement problem in a computing continuum as a Mixed Integer Non-Linear Problem (MINLP), and we propose a design-time tool for its efficient solution. We first propose a Random Greedy algorithm to minimize the cost of the placement while guaranteeing some response time performance constraints. Then, we develop some heuristic methods such as Local Search, Tabu Search, Simulated Annealing and Genetic Algorithms, to improve the initial solutions provided by the Random Greedy. To evaluate our proposed approach, we designed an extensive experimental campaign, comparing the heuristics methods with one another and then the best heuristic against Best Cost Performance Constraint (BCPC) algorithm, a state-of-the-art approach. The results demonstrate that our proposed approach finds lower-cost solution than BCPC (27.6% on average) under the same time limit in large-scale systems. Finally, during the validation in a real edge system including FaaS resources our approach finds the globally optimal solution, suffering a deviation of around 12% between actual and predicted costs.

Index Terms—**Cloud Edge computing, AI services, Component placement, Resource selection.**

1 Introduction

Artificial Intelligence (AI) applications and Deep Learning (DL) are now pervasive. IDC expects overall worldwide AI Services spending to reach $52.6 billion by 2025, at a CAGR of 21.9% [1]. The Cloud Computing paradigm supports the AI development, but cloud resources are far from end users and Internet of Things (IoT) devices producing data, which causes long delays. To better support novel AI and big-data applications, the computation is migrating towards mobile computing and IoT: having the computing resources at the periphery of the network reduces latency and bandwidth requirements while increasing energy efficiency and privacy protection. On the other hand, the limited capacity of edge devices may prevent them from executing AI applications requiring heavy processing, particularly under Quality of Service (QoS) response time constraints. Therefore, in the emerging computing continuum edge and cloud computing cooperate for the optimal deployment of AI applications, whose components are distributed among devices with highly heterogeneous capacities [2]–[4]. Moreover, since AI applications frequently include Deep Neural Networks (DNNs) that require a large computational capacity, partitioning the DNN components is a common approach to optimize the resources usage (see, e.g., [5], [6]).

In computing continua, the fragmentation and heterogeneity of the available devices (both in terms of computational power and execution costs) make the resource selection and component placement problems very challenging. In this paper, we propose *SPACE4AI-D* (System PerformAnce and Cost Evaluation on Cloud for AI applications Design), a tool to tackle these problems at design-time, where a user is allowed to define multiple candidate resources. Given an expected load, SPACE4AI-D selects the resources that will be used in production to serve and AI inference pipeline with the goal of minimizing the cost and fulfilling the performance constraints. Each AI application component is assumed to be a single DNN, which might be partitioned differently, giving rise to many candidate deployments.[1]

SPACE4AI-D develops an efficient Random Greedy (RG) algorithm, with the goal of identifying the minimum-cost placement across heterogeneous resources, including edge devices, cloud GPU-based VMs and FaaS configurations, under QoS response time constraints. Furthermore, the tool includes several heuristic algorithms, i.e., Local Search (LS), Tabu Search (TS), Simulated Annealing (SA) and Genetic Algorithms (GAs), implemented to enhance the RG solution.

To the best of our knowledge, this is the first work that

• *H. Sedghani, F. Filippini and D. Ardagna are with Politecnico di Milano. Milan, Italy. Email: {name.lastname}@polimi.it (Corresponding author: Hamta Sedghani, Email: hamta.sedghani@polimi.it).*

1. In the following we will use *partition placement* and *component placement* alternatively.

considers alternative deployments for the AI applications components, including different DNN partitions and different resource candidates, with the goal of selecting the optimal deployment and resources while guaranteeing the QoS performance constraints.

This paper extends our previous work [3], introducing the following main contributions:

1) A Mixed Integer Non-linear Program to model the joint resource selection and components placement problem.
2) We consider both Machine Learning-based models and queuing theory to predict the performance of the DNN components and we tackle the optimization problem by proposing an efficient RG algorithm and several solution-enhancing heuristics (ranging from LS to GAs).
3) We compare the performance of the proposed algorithms with the Best Cost Performance Constraint (BCPC) state-of-the-art algorithm proposed in [7]. Experimental results show that our heuristics yield a significant cost reduction, up to 80% with respect to [7].
4) We validate our results on a real system including Rasperry Pi, Odroid devices, private VMs at the edge, Amazon EC2 VMs and AWS Lambda functions in the cloud. The deviations between actual and predicted cost and time are around 12% and 20%, 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 describes a running example to illustrate how our approach can be applied in a real scenario. Section 4 reports the mathematical formulation of our problem, while Section 5.1 and Section 5.2 introduce RG and the other heuristic algorithms we propose to tackle it. The experimental results are presented in Section 6. Related works are discussed in Section 7, while conclusions are finally drawn in Section 8.

2 Application and Resource Models

In this section, we introduce the general model developed for our resource selection 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. Sections 2.4 and 2.5 introduce the network model and the description of the system costs, respectively. For the reader's convenience, all the problem parameters and decision variables introduced in the next sections are summarized in Appendix in Table 3 and 4.

It is worth noting we assume that, as in other literature approaches (see, e.g., [8]–[13]), the input parameters (e.g., data exchange size among AI pipeline components, initial estimates of individual component service demands, network bandwidth) are known and fixed at nominal values, without considering the influence of data uncertainties, which is beyond the scope of this paper. The dynamic nature of some parameters, such as workload fluctuations, pertains to runtime problems, which we have discussed in detail in [12]. However, making correct decisions at design time is critical for addressing runtime problems, as incorrect resource selection may prevent runtime tools from effectively identifying feasible re-configurations, particularly for edge devices with limited capacity that often become performance bottlenecks if the input workload increases. Therefore, we consider a fixed peak input exogenous workload at design time.

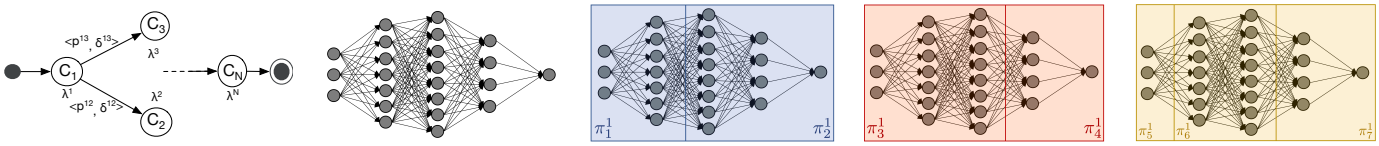
2.1 Application components model

As in other literature works ([14]–[17], we model each application as a directed acyclic graph (DAG), see Figure 1a, whose nodes represent the application components. We assume that each component is a Python function which can run in a Docker container deployed in an edge device, in a cloud Virtual Machine (VM) or according to the Function as a Service (FaaS) paradigm. For the sake of simplicity and according to other literature proposals [18]–[20], 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. Moreover, we consider DAGs including only sequential execution and branches, since, as in [18], we assume that loops are unfolded (or peeled) while parallel execution is not supported in this work. We denote the set of all application components by $\mathcal{I}$. The directed edge from node i to node k is labelled 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 $p^{ki} > 0$). If $Prec^i$ is the set of all components k executed immediately before i, then:

$$\lambda^i = \sum_{k \in Prec^i \subseteq \mathcal{I}} p^{ki} \lambda^k. \tag{1}$$

AI applications components are often Deep Neural Networks (DNNs), which might be partitioned differently according to resources capacity and network settings. Thus, we characterize the components by multiple *candidate deployments*, where a deployment denotes a set of different DNN partitions. Each partition can run on a device independently. We denote by $\mathcal{C}^i$ the set of all candidate deployments for component $i \in \mathcal{I}$. Each element $c_s^i \in \mathcal{C}^i$ is defined as $c_s^i = \{\pi_h^i\}_{h \in \mathcal{H}_s^i}$, where π_h^i denotes a DNN partition. The set $\mathcal{H}_s^i$ is defined as the set of indices h of all the partitions π_h^i in the candidate deployment c_s^i. An example of AI application component (denoted by $i = 1$) with its candidate deployments is reported in Figure 1. Three alternative deployments are available: c_1^1 and c_2^1, characterized by two partitions and denoted by $c_1^1 = \{\pi_1^1, \pi_2^1\}$ and $c_2^1 = \{\pi_3^1, \pi_4^1\}$, respectively, and c_3^1, characterized by three partitions and denoted by $c_3^1 = \{\pi_5^1, \pi_6^1, \pi_7^1\}$. In this setting, we will define $\mathcal{H}_1^1 = \{1, 2\}$, $\mathcal{H}_2^1 = \{3, 4\}$, and $\mathcal{H}_3^1 = \{5, 6, 7\}$. Without the loss of generality, generic Python code not running an AI inference function (implementing, e.g., intermediate data transformation or processing) can be modeled with a single deployment including a single partition.

Alongside p^{ik} (which is related to the transition from component i to component k), we introduce a parameter $\tilde{p}_{h\xi}^i$ to define the probability that partition π_ξ^i is executed just after partition π_h^i. Mechanisms as early stopping [5] entail that not all the partitions in a deployment are necessarily executed, which dictates the necessity of defining the probability of actually move from one to the other. Similarly, we



(a) DAG of components. (b) Component 1 (c) Candidate c_1^1 (d) Candidate c_2^1 (e) Candidate c_3^1

Figure 1: Example of AI application component DAG with its candidate deployments

define $\tilde{\delta}^i_{h\xi}$ as the size of data transferred from π^i_h to π^i_ξ, and we introduce the load $\tilde{\lambda}^i_h$ as:

$$\tilde{\lambda}^i_h = \begin{cases} \sum_{\pi^i_\xi \in Prec^i_h} \tilde{p}^i_{\xi h} \tilde{\lambda}^i_\xi & \text{if } Prec^i_h \neq \emptyset \\ \lambda^i & \text{otherwise,} \end{cases} \quad (2)$$

where $Prec^i_h$ denotes the set of all partitions preceding π^i_h, i.e., the set of all partitions π^i_ξ such that $\tilde{p}^i_{\xi h} > 0$. Note that, due to the definition of deployment, $Prec^i_h$ is either a singleton set or it is empty, if π^i_h is the very first partition in component i. In the latter case, $\tilde{\lambda}^i_h$ corresponds to the component load λ^i.

Finally, each partition π^i_h of component $i \in \mathcal{I}$ is, characterized by the memory requirement $\tilde{m}^i_h$ (expressed in MB).

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 expected response time. This includes the sum of the response times $\tilde{R}^i_h$ of all partitions π^i_h in the chosen deployment, plus additional terms due to network transmissions, as will be detailed in Section 2.4 and Section 4.2.

We define execution paths as sequences of application components, going 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* 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:

$$\mathcal{LC} = \{\langle i, \overline{LR}_i \rangle : \ i \in \mathcal{I}, \ R^i \leq \overline{LR}_i\}. \quad (3)$$

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:

$$\mathcal{GC} = \{\langle P, \overline{GR}_P \rangle : \ P \subseteq ep, \ ep \in \mathcal{EP}, \ \widehat{R}_P \leq \overline{GR}_P\}. \quad (4)$$

2.3 Resources general model

Computing continuum resources include devices located in the edge, cloud Virtual Machines (VMs) and Function as a Service (FaaS) configurations. FaaS is a category of cloud computing services where functions are executed transparently in Docker containers [21]. According to the underlying container status, there are two types of requests in FaaS: *hot requests* and *cold requests*. When a user invokes a function, the FaaS paradigm keeps the container running for a certain expiration time after the function execution (this time is provider-specific: according to the experiments performed in [22], the expiration time is about 10 minutes in AWS Lambda, Google Cloud Function and IBM Cloud Functions, and about 20 minutes for Azure Functions). If another request is raised before the shutdown, it is served instantaneously (hot request). When there are no hot containers available for the new incoming request (cold request), the provider has to bring up a new container, which causes a delay between the function invocation and execution (however, users do not pay for the resources during the delay).

We define different computational layers, including edge or cloud resources. The first layer includes local devices generating data (such as drones, as in the use case discussed in Section 3). The second layer is usually located at the edge and includes more powerful resources as smartphones, PCs or edge servers. For what concerns cloud resources, for the sake of simplicity, we assume that all the VMs come from a single cloud provider catalogue. The VMs selected at a given layer are homogeneous and evenly share the workload due to the execution of one or multiple partitions. If VMs with GPUs are available, we limit our selection to those characterized by a single GPU. Indeed, according to the current pricing models [23]–[25] costs and inference performance scale linearly with the number of GPUs, thus relying on such configurations improves the whole system availability. Finally, we include all FaaS configurations in the same layer, because each function runs on its own container independently. Each partition can be executed on FaaS configurations with different memory settings.

We denote the set of the candidate edge nodes by $\mathcal{J}_\mathcal{E} = \{1, ..., E\}$, the set of the candidate VM types in the cloud provider catalogue by $\mathcal{J}_\mathcal{C} = \{E+1, ..., E+C\}$, and the set of all FaaS configurations by $\mathcal{J}_\mathcal{F} = \{E+C+1, ..., E+C+F\}$. Consequently, the set of all candidate computing continuum resources will be denoted as $\mathcal{J} = \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{F}$.

For each Edge or VM resource $j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$, we denote by $n_j \in \mathbb{N}$ the total number of instances that can be selected in the placement. Moreover, for each partition π^i_h in all the candidate deployments c^i_s characterizing component $i \in \mathcal{I}$, we need to determine the compatibility with the available resources, i.e., we need to determine if a device $j \in \mathcal{J}$ can be used to execute π^i_h. To achieve this, we introduce a compatibility matrix $\mathcal{A} = [a^i_{hj}]$ defined as follows:

$$a^i_{hj} = \begin{cases} 1, & \text{if partition } \pi^i_h \text{ can run on node } j \in \mathcal{J} \\ 0, & \text{otherwise.} \end{cases} \quad (5)$$

2.4 Network model

Communications in the computing continuum are enabled by a set $\mathcal{D}$ of network domains connecting edge devices with each other and with the remote cloud back-end. Each $d \in \mathcal{D}$ is associated with a given technology, characterized by the access time a^d and the bandwidth B^d. Resource layers are included in, possibly, multiple network domains.

For each network domain d, we define $\mathcal{NL}^d = \{l_1, \cdots, l_{n^d}\}$ 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 partitions executed on different devices belonging to the same network domain $d \in \mathcal{D}$, depending on a^d, B^d and the amount of transferred data (see, e.g., [8]). Moreover, the network transfer time between consecutive components is needed to correctly evaluate the global execution time of a path, as detailed in Section 4.3.

Note that, given the gap between edge communication and cloud back-ends network performance, we can usually neglect the network delay in cloud since all VMs and all function instances are executed in the same data center.

2.5 System costs

The computing continuum resources are characterized by different costs. The costs of edge devices are amortized costs estimated for the single run of the target application (considering, e.g., the sum of the investment cost amortized along the lifetime horizon of the device, and the yearly management cost divided by the number of times the application is executed over a year). In the cloud, we consider hourly costs for the VMs, while FaaS costs are expressed in GB-second (see the AWS and Azure pricing models [24], [25]) and depend on the memory size, the functions duration, and the total number of invocations.

When the FaaS paradigm is considered, an additional cost might be incurred for each invocation: some providers (see, e.g., *AWS Step Functions* [25] and *Azure Logic Apps* [24]) offer a serverless orchestration service that allows users to combine AWS Lambda functions to build their business applications. In that case, a *state transition cost* is introduced to account for the message passing and coordination between two successive functions. Note that, with some third-party frameworks (e.g., *SCAR* [26] or *OSCAR* [27]) the orchestration can be supported by an architectural component avoiding transition costs.

3 A RUNNING EXAMPLE

To motivate our work, we investigate a use case related to the maintenance and inspection of a wind farm where the damages in wind turbines blades is identified in a computing continuum, based on images collected by drones. The application software is characterized by multiple components, consisting of DNNs that can be executed locally (on the drone, or on operators' PCs, or on local edge servers in the operators' van) or remotely in the cloud (in a VM or through FaaS). The set of components is illustrated in Figure 2 and can be executed overall on four layers. The dotted arrows connecting each component to the different resources denote the corresponding compatibility (i.e., they correspond to the elements of the compatibility matrix).

As an initial step, a drone with entry level or mid-range computation board (which configuration to buy is a decision that is taken by SPACE4AI-D), controlled remotely by a human operator, takes pictures of the wind turbine. Images are processed in batches and define the incoming workload λ. Each batch of pictures is subject to a first *exposure check*, identified in Figure 2 as component C_1, which determines if the image quality is sufficient for further processing. If not, the component, which is executed directly on the drone, triggers the acquisition of new images.

All pictures that pass the first check are subject to a second test, which is the preliminary analysis of the images (e.g., identifying if it is a part of the blade or not), implemented by C_2, while C_3 identifies the parts and the position of the image elements on the blade. These components require a significant amount of computing and storage power, but executing them at the edge may help in reducing the time needed to complete the maintenance and inspection process, if further data are needed to better identify the damages. C_4 is denoted as *damage-free check* and, as the name implies, checks if the inspected part is damaged or not (and in that case the computation can stop). Finally, C_5 is responsible for classifying the damage. Since these two last components need heavy computation, they are always performed in the cloud. Cloud resources are based on VMs and FaaS. The four computational layers belong, as described in Section 2.4, to three different network domains. In particular, drones and all edge resources communicate through a Wi-Fi network. Virtual Machines and the FaaS resources in the cloud are connected via a fiber optic network, while information are transferred from edge to cloud resources through a 5G network.

All components can, in principle, be characterized by multiple candidate deployments. E.g., for component C_4 we consider two deployments as in Figure 3: c_1^4 and c_2^4 are both characterized by two network partitions, with different computational loads and early-stopping probabilities. We assume that, in both cases, the first partition is preferably executed on edge resources, while the second in the cloud. Moreover, the early-stopping probability in the second deployment will be higher, since, in that case, most of the computation is executed in the first partition. If no damages are identified, computations will solely occur at the edge.

4 PROBLEM FORMULATION

This section presents the mathematical formulation of our problem: Section 4.1 introduces decision variables and placement constraints; the performance model is described in Section 4.2; Section 4.3 illustrates how we model the network transfer time, and QoS constraints are defined in Section 2.2. The system costs are modeled in Section 2.5, and the complete model is finally presented in Section 4.6.

4.1 Decision variables and allocation constraints

The resource selection and component placement problem we tackle in this paper is formulated as a Mixed Integer Non-Linear Programming (MINLP) optimization problem, aiming at minimizing the deployment cost at design time, while satisfying local and global QoS requirements for an AI inference pipeline.

We define the assignment decisions (i.e., we characterize which resource is selected at each computational layer, which candidate deployment is selected for each component and how the corresponding partitions are assigned to the available devices), by introducing the binary variables:

$$x_j = \begin{cases} 1 & \text{if node } j \in \mathcal{J} \text{ is used} \\ 0 & \text{otherwise,} \end{cases} \tag{6}$$

$$z_s^i = \begin{cases} 1 & \text{if deployment } c_s^i \text{ is selected for component } i \in \mathcal{I} \\ 0 & \text{otherwise,} \end{cases} \tag{7}$$

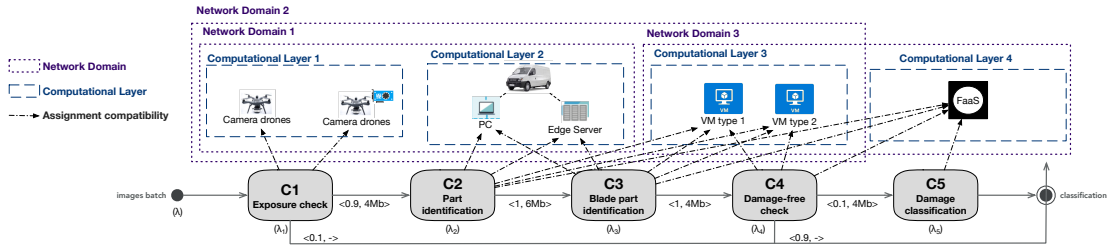
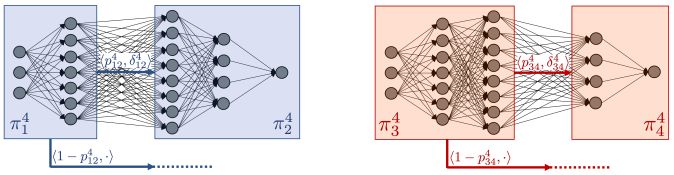


Figure 2: A Use case of identifying wind turbines blade damage.



(a) Deployment c_1^4 (b) Deployment c_2^4

Figure 3: Candidate deployments of component C_4

$$y_{hj}^i = \begin{cases} 1 & \text{if partition } \pi_h^i \text{ is deployed on node } j \in \mathcal{J} \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Note that exactly one deployment c_s^i must be selected for each component i, which can be expressed as follows:

$$\sum_{s:c_s^i \in \mathcal{C}^i} z_s^i = 1 \qquad \forall i \in \mathcal{I}. \tag{9}$$

Moreover, only one resource can be selected at each physical and virtual layer, namely:

$$\sum_{j \in \mathcal{L}^l} x_j = 1 \qquad \forall\, l : \mathcal{L}^l \subseteq \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}. \tag{10}$$

Once a deployment c_s^i is selected, we can assign the corresponding partitions to the most suitable resource. To assess the compatibility of the selected device, we prescribe:

$$y_{hj}^i \leq a_{hj}^i \qquad \forall i \in \mathcal{I},\ \forall h \in \mathcal{H}_s^i,\ \forall s : c_s^i \in \mathcal{C}^i. \tag{11}$$

Moreover, the assignment variable y_{hj}^i can be raised to one (i.e., a given partition π_h^i can be assigned to a device j) only if the partition is included in the deployment selected for component i. This can be prescribed by setting:

$$y_{hj}^i \leq z_s^i \qquad \forall i \in \mathcal{I},\ \forall h \in \mathcal{H}_s^i,\ \forall s : c_s^i \in \mathcal{C}^i. \tag{12}$$

Finally, each partition must be assigned to a single resource, which can be expressed by the following constraints:

$$\sum_{j \in \mathcal{J}} y_{hj}^i = 1 \qquad \forall i \in \mathcal{I},\ \forall h \in \mathcal{H}_s^i,\ \forall s : c_s^i \in \mathcal{C}^i. \tag{13}$$

Note that the maximum number of partitions that can be co-located in each device j is limited by the available memory M_j. Let $\tilde{m}_h^i$ be the memory required by partition π_h^i, as described in Section 2.1. The total memory requirement on device $j \in \mathcal{J}$ can be defined by summing the contribution of all partitions possibly executed on j, which entails:

$$\sum_{i \in \mathcal{I}} \sum_{s:c_s^i \in \mathcal{C}^i} \sum_{h \in \mathcal{H}_s^i} \tilde{m}_h^i y_{hj}^i \leq M_j \qquad \forall j \in \mathcal{J}, \tag{14}$$

Finally, requests cannot move back from cloud to edge. This means that, if a partition π_h^i is executed on cloud, the next partitions π_ξ^k in all the subsequent components must not be executed on edge. Thus, we impose:

$$\tilde{p}_{h\xi}^i y_{hj}^i \leq \sum_{v \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{F}} y_{\xi v}^i \qquad \forall i \in \mathcal{I},\ \forall s : c_s^i \in \mathcal{C}^i,$$
$$\forall\, h, \xi \in \mathcal{H}_s^i,\ \forall j \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{F} \tag{15}$$

$$p^{ik} y_{hj}^i \leq \sum_{v \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{F}} y_{\xi v}^k \ \forall i, k \in \mathcal{I}, s : c_s^i \in \mathcal{C}^i,$$
$$\sigma : c_\sigma^k \in \mathcal{C}^k,\ z_\sigma^k = 1,\ h \in \mathcal{H}_s^i, \xi \in \mathcal{H}_\sigma^k,\ j \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{F} \tag{16}$$

Constraints (15) enforce that, for each component, if a partition is allocated on Cloud, the subsequent partitions do not run on an Edge device. Constraints (16) enforce that, if a component has a partition running on Cloud, all the partitions of the next components must run on Cloud.

4.2 Performance Model

This section discusses the performance models that we used to characterize the response time of partitions executed on different resource types. Specifically, response times on edge nodes and cloud VMs (Section 4.2.1) can be modeled by relying on the queuing theory (as in other Edge-Cloud systems research works[8]–[11]) or estimated through Machine Learning (ML) models that adopt component load, the total number of used cores or some variants as a features (see [28]–[30]). For FaaS configurations, we rely on an external tool proposed in [31] (see Section 4.2.2).

4.2.1 Edge Nodes and Cloud Virtual Machines

For each computational layer l including edge resources or cloud VMs, we denote by D_{hj}^{il} the demanding time to run a partition π_h^i on a node $j \in \mathcal{L}^l \subseteq \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$, i.e., the average time required to run π_h^i on node j without resource contention (a single request to execute component i is served by the node, see [32]). As mentioned in Section 2.3, we denote by n_j the maximum number of instances of device j that can be selected in the placement (e.g., associated with reserved instances [24], [25], [33], [34]), and we introduce a variable $\widehat{y}_{hj}^i$ to count the number of instances assigned to a partition π_h^i. Note that, by assumption, the load is shared evenly among all devices used to run a single partition. Moreover, we define $\bar{y}_j$ as follows:

$$\bar{y}_j = \max_{\forall\, i \in \mathcal{I},\ s:c_s^i \in \mathcal{C}^i,\ h \in \mathcal{H}_s^i} \left(\widehat{y}_{hj}^i \right) \qquad \forall j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}. \tag{17}$$

If, as in [8], each edge node and cloud VM type is modeled as single server multiple class queue system (i.e., as an individual M/G/1 queue), the response time of a partition π_h^i possibly deployed at the edge or cloud in any device $j \in \mathcal{L}^l \subseteq \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$ can be computed as:

$$\tilde{R}_h^i = \sum_{l:\mathcal{L}^l \subseteq \mathcal{J}_\mathcal{E}, j \in \mathcal{L}^l} \left(\frac{D_{hj}^{il} y_{hj}^i}{1 - U_j} \right), \tag{18}$$

where U_j is the utilization of device j, defined as:

$$U_j = \frac{\sum_{k \in \mathcal{I}} \sum_{\sigma:c_\sigma^k \in \mathcal{C}^k} \sum_{\xi \in \mathcal{H}_\sigma^k} D_{\xi j}^{kl} y_{\xi j}^k \tilde{\lambda}_\xi^k}{\widehat{y}_{hj}^i}. \tag{19}$$

We further need to prescribe that, if any partition is executed on a device 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:

$$\widehat{y}_{hj}^i > 0 \implies U_j < 1, \tag{20}$$

which can be written as:

$$\begin{cases} \widehat{y}^i_{hj} + M_C(1 - y^i_{hj}) > 0 \\ \widehat{y}^i_{hj} + M_C(1 - y^i_{hj}) > \sum_{k \in \mathcal{I}} \sum_{\sigma : c^k_\sigma \in \mathcal{C}^k} \sum_{\xi \in \mathcal{H}^k_\sigma} \tilde{\lambda}^k_\xi D^{kl}_{\xi j} y^k_{\xi j} \\ \widehat{y}^i_{hj} \leq M_C y^i_{hj}, \end{cases} \tag{21}$$

where M_C is a "large enough" constant.

If, vice versa, the response time of a partition is estimated by relying on ML models, then $\tilde{R}^i_h$ can be expressed as a black box $\tilde{R}^i_h = f(\tilde{\lambda}^i_h, \widehat{y}^i_{hj})$, where $f(\cdot)$ is a non-linear regression function (see also [30] and Section 6.4).

4.2.2 Function as a Service

We denote by $d^{i,hot}_{hj}$ and $d^{i,cold}_{hj}$ the execution time of a hot and cold request for partition π^i_h on a FaaS configuration $j \in \mathcal{L}^l \subseteq \mathcal{J}_\mathcal{F}$, respectively. Moreover, we define the average execution time d^i_{hj} of partition π^i_h on function configuration $j \in \mathcal{L}^l \subseteq \mathcal{J}_\mathcal{F}$, which depends, as discussed in [31], on $d^{i,hot}_{hj}$, $d^{i,cold}_{hj}$, the expiration threshold and the arrival rate of function j, that is equal to the incoming workload $\tilde{\lambda}^i_h$ of the partition i running on the function.

According to these definitions, we can write the response time of every partition π^i_h possibly running at the function $j \in \mathcal{L}^l \subseteq \mathcal{J}_\mathcal{F}$ as follows:

$$\tilde{R}^i_h = \sum_{l : \mathcal{L}^l \subseteq \mathcal{J}_\mathcal{F}, j \in \mathcal{L}^l} d^i_{hj} y^i_{hj}, \tag{22}$$

where d^i_{hj} is computed relying on the tool proposed in [31]. In that research work, a continuous-time Semi-Markov Process is leveraged, which allows to achieve a percentage error of about 10-15% (which is accurate enough for design-time decisions [32]).

4.3 Network transfer time

As mentioned in Section 2.4, to evaluate the global execution time of a component $i \in \mathcal{I}$ we need to compute the network delay due to data transmissions between consecutive partitions. To estimate this, we introduce, for each pair of partitions (π^i_h, π^i_ξ) in component i, and for each network domain $d \in \mathcal{D}$, the binary variable $\tilde{w}^{id}_{h\xi}$, which is 1 if π^i_h and π^i_ξ are consecutive (i.e., $\tilde{p}^i_{h\xi} > 0$) and are executed on different devices in the same network domain d.

Thus, we obtain the list of network domains that can support the data transmission between the two partitions:

$$\widetilde{W}^i_{h\xi} = \{d \in \mathcal{D} : y^i_{hj} = 1,\ y^i_{\xi r} = 1,\ \forall\, j, r \in \mathcal{UL}^d\}, \tag{23}$$

where $\mathcal{UL}^d$ is the set of resource indexes included in the network domain $d \in \mathcal{D}$, namely:

$$\mathcal{UL}^d = \bigcup_{l \in \mathcal{ND}^d} \mathcal{L}^l. \tag{24}$$

Note that $\widetilde{W}^i_{h\xi}$ is empty when the resources assigned to the two partitions do not have any shared network domain, otherwise it contains at least one domain (if the partitions assignment are compatible according to matrix $\mathcal{A}$). The network domain exploited for the communication is denoted by the variable $\hat{w}^i_{h\xi}$, defined as:

$$\hat{w}^i_{h\xi} = \begin{cases} \underset{d \in \widetilde{W}^i_{h\xi}}{\operatorname{argmin}} \left(a^d + \dfrac{\tilde{\delta}^i_{h\xi}}{B^d} \right) & \text{if } \widetilde{W}^i_{h\xi} \neq \emptyset \\ 0 & \text{otherwise,} \end{cases} \tag{25}$$

Since there is not any network delay between two partitions running on the same resource, we define binary variable $f_{h\xi}$ to characterize if two partitions π^i_h and π^k_ξ are located in the same resource as follows:

$$f_{h\xi} = \begin{cases} 1 & \text{if } y^i_{hj} = 1,\ y^k_{\xi r} = 1, j \neq r, \\ & \forall i, k \in \mathcal{I}, s : c^i_s \in \mathcal{C}^i, \sigma : c^k_\sigma \in \mathcal{C}^k, \\ & h \in \mathcal{H}^i_s, \xi \in \mathcal{H}^k_\sigma, \forall j, r \in \{\mathcal{J_E} \cup \mathcal{J_C}\} \\ 0 & \text{otherwise,} \end{cases}$$

where i and k can be the same or different components. This allows us to express the network delay between the pair of partitions (π^i_h, π^i_ξ) such that $\tilde{p}^i_{h\xi} > 0$ as:

$$\tilde{t}^i_{h\xi} = \begin{cases} \left(a^{\hat{w}^i_{h\xi}} + \dfrac{\tilde{\delta}^i_{h\xi}}{B^{\hat{w}^i_{h\xi}}} \right) f_{h\xi} & \text{if } \hat{w}^i_{h\xi} \neq 0 \\ \infty & \text{otherwise.} \end{cases} \tag{26}$$

Therefore, taking into account early-stopping probabilities of the partitions, the overall intra-component expected delay for component i is:

$$t^i = \left[\sum_{n=1}^{|\mathcal{H}^i_s|-1} \left(\prod_{j=1}^{n} \tilde{p}^i_{k_j k_{j+1}} \right) \tilde{t}^i_{k_n k_{n+1}} \right]_{\{k_u\}_{u=1}^{|\mathcal{H}^i_s|} = \mathcal{H}^i_s,\ z^i_s = 1}. \tag{27}$$

On the other hand, to compute the response time of a path P, we also need to define the network transfer time due to data transmissions between consecutive components. Note that the data transmissions between component i and k happen between the last executed partition in component i (which vary due to early stopping) and the first running partition of component k.

As before, indicating with π^i_h the partitions of component i and with π^k_ξ the first partition of component k, we define the list of network domains that can support the data transmission between the two components as:

$$W^{ik}_{h\xi} = \{d \in \mathcal{D} : y^i_{hj} = 1,\ y^k_{\xi r} = 1 \wedge \alpha^k_\xi = 1,\ \forall j, r \in \mathcal{UL}^d\}. \tag{28}$$

We define the variable $w^{ik}_{h\xi}$ as:

$$w^{ik}_{h\xi} = \begin{cases} \underset{d \in W^{ik}_{h\xi}}{\operatorname{argmin}} \left(a^d + \dfrac{\delta^{ik}}{B^d} \right) & \text{if } W^{ik}_{h\xi} \neq \emptyset \\ 0 & \text{otherwise,} \end{cases} \tag{29}$$

and characterize the network delay between all the running partitions of component i and the first partition of component k as:

$$t^{ik}_{h\xi} = \begin{cases} \left(a^{w^{ik}_{h\xi}} + \dfrac{\delta^{ik}_{h\xi}}{B^{w^{ik}_{h\xi}}} \right) f_{h\xi} & \text{if } w^{ik}_{h\xi} \neq 0 \\ \infty & \text{otherwise.} \end{cases} \tag{30}$$

Eventually, the overall expected transmission delay between component i and component k reads:

$$t^{ik} = \Bigg[(1 - \tilde{p}^i_{v_1 v_2}) t^{ik}_{v_1 \xi} + \sum_{n=2}^{|\mathcal{H}^i_s|} \left(\prod_{j=1}^{n-1} \tilde{p}^i_{v_j v_{j+1}} (1 - \tilde{p}^i_{v_n v_{n+1}}) \, t^{ik}_{v_n \xi} \right) \Bigg]_{\{v_n\}_{n=1}^{|\mathcal{H}^i_s|} = \mathcal{H}^i_s,\ z^i_s = 1\ \alpha^k_\xi = 1}. \tag{31}$$

4.4 Local and global QoS constraints

As mentioned in Section 2.3, given the definitions in Equations (18) and (22), as well as the network transfer time defined in Equation (31), we can compute the response time R^i of a specific component $i \in \mathcal{I}$ as:

$$R^i = \left[\tilde{R}^i_{k_1} + \sum_{n=1}^{|\mathcal{H}^i_s|-1} \left(\prod_{j=1}^{n} \tilde{p}^i_{k_j k_{j+1}} \right) \tilde{R}^i_{k_{n+1}} \right]_{\substack{\{k_n\}_{n=1}^{|\mathcal{H}^i_s|} \in \mathcal{H}^i_s, \\ z^i_s = 1}} + t^i. \tag{32}$$

The first term corresponds to the execution time of the first partition in the selected deployment, while the second one represents the total transfer time among such partitions.

If we consider any path P, i.e., any sequence of components, the relative response time $\widehat{R}_P$ can be defined as follows:

$$\widehat{R}_P = \sum_{i \in P} R^i + \sum_{i,k \in P: i \neq k} t^{ik}, \tag{33}$$

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. Note that, contrary to the approach adopted for the components, we do not compute the expected path response time by using components execution probabilities. Instead, we define $\widehat{R}_P$ as the algebraic sum of the response time of the components forming the path. This yields a conservative estimate of the actual path response time.

In order to guarantee QoS requirements of the application, we can define global response time constraints for the different paths, as mentioned in Section 2.1. This means setting a threshold $\overline{GR}_P$ and prescribing:

$$\widehat{R}_P \leq \overline{GR}_P \qquad \forall \langle P, \overline{GR}_P \rangle \in \mathcal{GC}. \tag{34}$$

We can also prescribe local constraints thresholds for the response times of single components, as follows:

$$R^i \leq \overline{LR}_i \qquad \forall \langle i, \overline{LR}_i \rangle \in \mathcal{LC}. \tag{35}$$

4.5 System costs

As introduced in Section 2.5, edge devices are characterized by the estimated amortized costs for the single run of the target application, cloud VMs are characterized by hourly costs, and FaaS configurations costs are expressed in GB-second. To compute them, we denote by T the overall time an application is active for a single run, and we assume that T is less than or equal to one hour.

The execution cost of all edge devices is defined as:

$$C_E = \sum_{j \in \mathcal{J}_\mathcal{E}} c_j^E x_j, \tag{36}$$

where c_j^E is the amortized cost of the edge device $j \in \mathcal{J}_\mathcal{E}$, while the total execution cost on VMs can be computed as:

$$C_C = \sum_{j \in \mathcal{J}_\mathcal{C}} c_j^C \bar{y}_j, \tag{37}$$

where c_j^C and $\bar{y}_j$ denote the hourly cost and the maximum number of running VMs of type $j \in \mathcal{J}_\mathcal{C}$, respectively.

Let $c_{hj}^{F,i}$ be the GB-second unit cost for executing π_h^i on the FaaS configuration $j \in \mathcal{J}_\mathcal{F}$. FaaS total costs depend on memory size, functions duration, and total number of invocations. The execution cost of the FaaS layer is:

$$C_F = \sum_{i \in \mathcal{I}} \sum_{s: c_s^i \in \mathcal{C}^i} \sum_{h \in \mathcal{H}_s^i} \sum_{j \in \mathcal{J}_\mathcal{F}} c_{hj}^{F,i} d_{hj}^{i,hot} y_{hj}^i \tilde{\lambda}_h^i T. \tag{38}$$

Note that the cost of the used memory is embedded in $c_{hj}^{F,i}$. Indeed, as mentioned, $d_{hj}^{i,hot}$ is inversely proportional to the memory $\tilde{m}_h^i$ allocated to partition π_h^i (see [7]).

According to some FaaS providers (see, e.g., [24], [25]), we need to introduce a *state transition cost*, denoted here with c^T, to model the additional charge for the message passing and coordination between two successive functions. If, however, the orchestration is supported by an architectural component (see, e.g., [26], [27]), this cost is set to $c^T = 0$. Without loss of generality, we can thus formulate the transition cost as:

$$C_T = \sum_{i \in \mathcal{I}} \sum_{s: c_s^i \in \mathcal{C}^i} \sum_{h \in \mathcal{H}_s^i} \sum_{j \in \mathcal{J}_\mathcal{F}} c^T y_{hj}^i \tilde{\lambda}_h^i T. \tag{39}$$

Given that the size of tensors between layers is usually small (e.g., in the widely adopted YOLOv5 it is at most 6MB [13]), we assume the cloud provider cost for the data transfer is negligible. However, our model can be easily extended to account for data transfer costs. This extension would involve incorporating a cost for the network domains that would be non-zero only for wide area network connections (e.g., 5G) and would include network and cloud provider data transfer costs.

4.6 Optimization problem

The optimization model proposed in this work aims at describing an edge/cloud component placement problem for an AI inference pipeline, whose objective is the minimization of the total application execution costs.

The Mixed Integer Non-Linear Programming (MINLP) formulation of the problem reads:

$$\min C_E + C_C + C_F + C_T \tag{P1a}$$

Subject to:

(9), (11), (12), (13), (14), (15), (16) (21), (34), (35) and

$$y_{hj}^i \leq x_j \qquad \forall\, i \in \mathcal{I},\ s : c_s^i \in \mathcal{C}^i,\ h \in \mathcal{H}_s^i,\ j \in \mathcal{J}, \tag{P1b}$$

$$\widehat{y}_{hj}^i \leq \bar{y}_j \qquad \forall\, i \in \mathcal{I},\ s : c_s^i \in \mathcal{C}^i,\ h \in \mathcal{H}_s^i,\ j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}, \tag{P1c}$$

$$y_{hj}^i \leq \widehat{y}_{hj}^i \qquad \forall\, i \in \mathcal{I},\ s : c_s^i \in \mathcal{C}^i,\ h \in \mathcal{H}_s^i,\ j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}, \tag{P1d}$$

$$\widehat{y}_{hj}^i \leq n_j y_{hj}^i \qquad \forall\, i \in \mathcal{I},\ s : c_s^i \in \mathcal{C}^i,\ h \in \mathcal{H}_s^i,\ j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}, \tag{P1e}$$

$$\bar{y}_j \leq n_j x_j \qquad \forall j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}, \tag{P1f}$$

$$\sum_{\substack{s: c_s^i \in \mathcal{C}^i \\ h \in \mathcal{H}_s^i}} \alpha_h^i = 1 \qquad \forall i \in \mathcal{I}, \tag{P1g}$$

$$\widehat{y}_{hj}^i, \bar{y}_j \in \mathbb{N} \qquad \forall i \in \mathcal{I},\ s : c_s^i \in \mathcal{C}^i,\ h \in \mathcal{H}_s^i,\ j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}, \tag{P1h}$$

$$y_{hj}^i, x_j, \alpha_h^i \in \{0,1\} \qquad \forall i \in \mathcal{I},\ s : c_s^i \in \mathcal{C}^i,\ h \in \mathcal{H}_s^i,\ j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}. \tag{P1i}$$

Constraints (P1b) bound the allocation of components to the devices. Constraints (P1c) bound the number of allocated VMs to the maximum number of running VMs of type j, while Constraints (P1d) avoid running a component on VMs that are not assigned to it. Constraints (P1e) guarantee that the number of VMs of type j assigned to a component does not exceed the maximum available number n_j, and Constraints (P1f) guarantee that the cluster size does not exceed the maximum number of available resources for the selected devices. Finally, Constraints (P1g) impose that only a single partition can be the *first running* in each component. Constraints (P1i) - (P1h) define the decision variables domain.

5 Heuristic Algorithms

Due to the QoS performance constraints, both if expressed through M/G/1 models or ML, our mathematical formulation is non linear. Moreover, it can be seen as an extension of [35] due to the presence of multiple alternative deployments for each component. Since the problem in [35] is NP-hard, also our problem is NP-hard.

To determine a good-quality solution that limits operational costs while meeting hardware, memory and QoS constraints, we propose several heuristic methods. They all share an initial *Random Greedy* phase (see Section 5.1) that iteratively constructs a pool of k candidate solutions.

Then, each method is characterized by a specific *solution enhancement* process, as detailed in Section 5.2.

5.1 The Random Greedy algorithm

The Random Greedy (RG) method (see Algorithm 1) receives as input the complete system description, including all components and resources and the corresponding characteristics, and a maximum running time (MT). At each iteration, it constructs a new candidate solution by: randomly selecting a resource j in each computational layer (lines 5–7) and a NN deployment c_s^i for each component $i \in \mathcal{I}$ (line 9); assigning each partition π_h^i of the chosen deployment to a randomly selected compatible resource (lines 10–12); randomly choosing the number of instances to consider for each Edge device and Cloud VM (lines 14–16). If the generated solution is feasible (i.e., it satisfies Constraints (P1b)–(P1f), (14), (15), (16) (35), (34) from Section 4.6), we tentatively reduce the number of Edge device and Cloud VM instances currently in use (lines 18–20) to limit the operational costs. When the loop terminates because the maximum algorithm runtime is reached, the method returns the k candidate solutions with lower cost (lines 25-27).

Algorithm 1 Random Greedy algorithm

1: **Input:** $\mathcal{J}_\mathcal{E}$, $\mathcal{J}_\mathcal{C}$, $\mathcal{J}_\mathcal{F}$, $\mathcal{L}$, $\mathcal{C}$, $\mathcal{H}$, $\mathcal{ND}$, A, DAG, $\mathcal{LC}$, $\mathcal{GC}$, $\widehat{c}$, $\bar{c}$, $\tilde{c}$, MT, k
2: **Initialization:** $Solutions \leftarrow \emptyset$, $StartTime \leftarrow CurrentTime$
3: **while** $CurrentTime - StartTime < MT$ **do**
4: $x \leftarrow [0], y \leftarrow [0], \widehat{y} \leftarrow [0]$
5: **for** $l \in \mathcal{L}$ **do**
6: Randomly pick a node $j \in \mathcal{L}^l$ and set $x_j \leftarrow 1$
7: **end for**
8: **for** $i \in \mathcal{I}$ **do**
9: Randomly pick a deployment $c_s^i \in \mathcal{C}^i$
10: **for** $h \in \mathcal{H}_s^i$ **do**
11: Randomly pick $j \in \mathcal{J}$ s.t. $x_j = 1 \wedge a_{hj}^i = 1$; set $y_{hj}^i \leftarrow 1$
12: **end for**
13: **end for**
14: **for** $j \in \{\mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}\}$ such that $x_j = 1$ **do**
15: $\widehat{y}_{hj}^i \leftarrow random[1, n_j] \cdot y_{hj}^i \forall i \in \mathcal{I}, \forall h \in \mathcal{H}_s^i$
16: **end for**
17: **if** $IsFeasible(\langle x, y, \widehat{y} \rangle)$ **then**
18: **for** $j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$ such that $x_j = 1$ **do**
19: $ReduceVMClusterSize(j)$
20: **end for**
21: $Solutions \leftarrow Solutions \cup \langle x, y, \widehat{y} \rangle$
22: **end if**
23: **end while**
24: **if** $Solutions \neq \emptyset$ **then**
25: Sort $Solutions$ by cost
26: **return** top k solutions of $Solutions$
27: **else**
28: No feasible solution found
29: **end if**

5.2 The solution enhancement process

The heuristic methods we propose improve the candidate RG solutions by exploring their neighborhoods, defined as detailed in Section 5.2.1, leveraging algorithms based on Local or Tabu Search (LS and TS, Section 5.2.2), Simulated Annealing (SA, Section 5.2.3) and Genetic Algorithm (GA, Section 5.2.4). In particular, LS, TS and SA adopt a multi-start approach considering the k best solutions returned by RG, while GA starts from a population proportional to k.

5.2.1 Generating neighbors

This section describes the list of *moves* we propose to explore the neighborhood of each candidate solution sol. We use the running example in Section 3 to clarify the understanding of generating neighbors, assuming that the initial RG solution assigns component C_1 to the first drone of CL_1, C_2 and C_3 on PC, C_4 on VM type 1 and C_5 on FaaS. In general, moves consist of defining a suitable set of candidate source nodes J_s and iterate over $j_s \in J_s$ applying the required changes until a new feasible solution sol' is found.

- *Change component placement:* J_s includes all the running nodes, sorted in increasing order of utilization and cost. Selected j_s, we migrate each partition π_h^i running on it to the least-used (or less expensive) compatible resource, chosen among those that are already running or that are hosted in a currently-inactive computational layer. In Figure 2, migrating C_2 or C_3 from PC to VM type 1 or 2 can generate four distinct candidate sol'. Note that although these moves might not be immediately beneficial, they could prove advantageous in subsequent alterations of the current solution.
- *Change resource type:* J_s includes the running Edge nodes and Cloud VMs, sorted as in the previous move. We tentatively substitute j_s with an alternative node type, selected among those compatible with all partitions running on j_s. In the running example, migrating both C_2 and C_3 from PC to VM type 1 or 2, thereby eliminating the use of PC, can generate two candidate neighbors for the initial solution.
- *Migrate to FaaS:* Defining J_s as in the previous move, we tentatively migrate all partitions running on j_s to the largest (in terms of memory and cost) compatible FaaS instance, if any. If all partitions can be moved, j_s is switched off in the new solution. In Figure 2, migrating C_4 from VM type 1 to FaaS can yield a beneficial solution, as FaaS is typically more cost-efficient than Cloud VMs.
- *Migrate from FaaS:* J_s corresponds to the set $\mathcal{J}_\mathcal{F}$ of FaaS configurations, and we tentatively migrate the partition running on j_s to a compatible Edge node or Cloud VM, considered in increasing order of utilization and cost.
- *Change FaaS instance:* Defining J_s as in the previous move, we select the destination node to migrate each π_h^i among the alternative (compatible) FaaS instances.

Note that, in all moves, we iteratively explore the nodes in J_s until the first feasible neighbor is found. The only exception is the last move, where we generate a new candidate sol' for each node in $J_s \equiv \mathcal{J}_\mathcal{F}$.

5.2.2 Local Search and Tabu Search

LS [36] is a solution-enhancement procedure that starts from an initial feasible solution and iteratively updates it by exploring its neighbors. To mitigate the risk of getting stuck in local minima, an *Iterated* version (ILS) can be implemented by considering a pool of initial solutions. Each neighborhood exploration is performed independently, thus the knowledge obtained during each LS phase is isolated and not used by the other ones. SPACE4AI-D implements ILS by considering the k candidate RG solutions, and explores the feasible neighborhoods exhaustively by applying the moves described in Section 5.2.1, picking the best neighbor for the next move. Our LS terminates when no feasible neighbor exists or a predefined maximum time threshold (MT) is reached, returning the best achieved result.

TS [37] is a metaheuristic that implements an iterative, memory-based neighborhood-search method that explores the solution space by replacing the current solution with the best not-yet-visited neighbor, aiming to overcome the limitations of local optimality. The most recent explored moves are kept in a fixed-size *tabu list* to guarantee that we do not loop twice over the same solution. SPACE4AI-D implements a random TS, generating the neighborhoods with the same moves of LS but randomly selecting one neighbor among all the generated ones instead of considering the best neighbor in each step. This allows to choose worse solutions and to avoid getting stuck in local optima. The procedure will stop when all the currently generated neighbors are already in the tabu list or the maximum time is over.

5.2.3 Simulated Annealing

SA [38] is a metaheuristic method to approximate the optimal solution, inspired by an annealing process: high temperatures turn the state unstable by providing the energy needed to break bonds, so that free atoms can move to new positions quickly. In turn, the cooling process leads to atoms stabilization and consequently to an equilibrium. In our implementation, we generate the solutions neighbors as in Section 5.2.1, and we initialize the temperature to a large number. When the temperature is high, we accept neighbors that worsen the cost to allow further exploration of the solution space. Progressively lowering the temperature, we select only neighbors that improve the cost. We stop when the temperature is small enough or we reach MT.

5.2.4 Genetic Algorithms

GAs [38] are metaheuristics inspired by biological evolution, applied to solve both constrained and unconstrained optimization problems through an intelligent exploitation of random search. The algorithm repeatedly modifies a population of solutions using *mutation* and *crossover* actions. At each iteration, GAs select solutions from the current population of *parents* and generate *children* for the next generation. To apply GA in SPACE4AI-D, we define genes and chromosomes as shown in Figure 4. Each gene G_i is a matrix that specifies the resource assignment of the partitions related to component i (see Figure 4a), i.e., it corresponds to the matrix $\widehat{y}^i_{hj}$ for a fixed i. A chromosome, which represents a solution ($\widehat{y}^i_{hj} \forall i$), consists of a list of all components (genes) in the application DAG (see Figure 4b).

	1	2	.....	$\lvert\mathcal{J}\rvert$
1	1	0	.....	0
2	0	1	.....	0
:	:	:	.....	:
:	:	:	.....	:
:	:	:	.....	:
$\lvert\mathcal{H}^i_1+\mathcal{H}^i_2+\ldots+\mathcal{H}^i_{\lvert e^i\rvert}\rvert$	0	0	.....	0

(a) Gene G_i.

G_1	G_2	....	$G_{\lvert\mathcal{I}\rvert}$

(b) Chromosome.

Figure 4: Definition of gene and chromosome in GA.

We define the mutation and crossover functions as:

- $Mutate$ compares a given $MutationRate$ with a random number $r \in (0, 1)$. If r is lower, we randomly select one of the implemented moves to generate sol'. Otherwise, we return sol without any modification.
- $Crossover$ receives two parent solutions (P_1 and P_2) and generates a partition point in the parents chromosomes. These are both split to obtain two children, mixing the first part of P_1 chromosome with the second part of P_2 chromosome, and vice versa. Due to some problem-specific constraints, such as selecting a single resource in each computational layer, mixing directly the chromosome parts will likely result in an infeasible solution. We designed Algorithm 2 (reported in Appendix) to preserve the feasibility throughout the mixing operation, making the search more effective.

As the other methods, GA stops when it reaches MT.

6 Experimental results

To evaluate the performance and scalability of SPACE4AI-D, we ran multiple experiments generating instances with both light and strict constraints. In particular, we extended the validation of our previous work [3], where we considered only sequential execution of application components, by introducing application DAGs with branches. Moreover, we validated our tool by considering a prototype application run in a real edge system and we compare our solution with the stat-of-the-art approach proposed in [7].

The setup of our experiments is described in Section 6.1. In Section 6.2, we present an ablation study where we compare the outcomes of the pure RG method with the results achieved by the heuristics implemented by considering the solution enhancement strategies presented in Section 5.2. Then, we compare our Local Search with the BCPC method [7] in Section 6.3. The validation in a real system is described in Section 6.4.

Note that considering the maximum algorithm runtime (MT) as stopping criterion instead of the maximum number of iterations increases the fairness of the comparison, since the execution time of a single iteration with different heuristics may significantly differ. To quantitatively compare a target method with a baseline, we define the *Cost ratio* as:

$$\text{Cost ratio} = \frac{(BaselineMethodCost - TargetMethodCost)}{TargetMethodCost}, \quad (40)$$

where the baseline methods are RG (run for MT) in Section 6.2 and the BCPC method in Section 6.3, respectively, while the target methods are the other heuristics (and RG running for $2MT$) in Section 6.2, and LS in Section 6.3. Note that a positive Cost ratio is achieved when the target method outperforms the baseline, since the latter yields a higher cost and we are facing a minimization problem.

All the experiments were run on a Linux Ubuntu server with 2.2 GHz CPU Intel with 40 cores and 32 GB memory.

6.1 Experiments setup

In our experiments, we randomly generated a set of instances representative of real systems. Similarly to our previous work [3] and to [7], we consider three different scenarios at different scales including application DAGs with 7, 10 and 15 components. We report the average Cost ratio and execution time achieved by considering 10 random instances with the same size. As in [7], we randomly generated the application DAG for every instance by relying on Networkx. We randomly selected between 1 and 3 deployments for each component and between 1 and 4 partitions for each deployment. We also considered at most 4 computational layers in edge, including up to 4 different node types each in the largest scenarios, and 3 computational layers in the cloud, with 5 different VM types

each and a maximum number of VM instances randomly selected between 3 and 5 for each type. We introduced up to 5 local and global constraints. Local and global constraints thresholds (expressed in seconds) related to light constraints are set randomly in the range of $[50, 100]s$ and $[200, 300]s$, respectively, while the same thresholds for strict constraints scenario are set in the range of $[7, 10]s$ and $[20, 25]s$. Further details are reported in Appendix in Table 5.

For serverless functions, we use the tool proposed in [31] to predict the performance of partitions running on FaaS. The number of requests per second (λ) is normalized between 0 and 1.

In the experiments comparing the heuristics (Section 6.2), the full DNN service demands were randomly generated in the range of $[1, 5]s$ for edge resources, $[0.5, 2]s$ for VMs (as in [19]), and $[2, 5]s$ for cold and warm FaaS requests (as in [39]). The service demands of partitions are proportional to the number of DNN layers within the partitions (see e.g., [13]) In all these experiments, we use M/G/1 queuing models to predict the performance of partitions running on the edge nodes and cloud VMs, which can be estimated with a percentage error of about 10% [8]. In Section 6.3, to obtain a fair comparison with the BCPC method, the service demands are set equal to those considered in [7]. For the evaluation in a real system (Section 6.4), we use ML models to predict the performance of components running on edge nodes and AWS cloud instances.

The same k best RG solutions ($k = 10$ at most, according to how many feasible candidates can be generated) were used as starting points by LS, TS and SA. The Tabu list size is set to 50. Initial temperature in SA is set to 5 and it is reduced exponentially by factor of 0.99 in each step to reach out to $\epsilon = 10^{-6}$. Since GA generates the solutions mostly based on random behaviour, it needs a larger number of initial random solutions; therefore, we used the $4k$ best RG solutions as initial population. The mutation and crossover rate are set to 0.7 and 0.5, respectively.

The code used to run our experiments and all the input and output files are available as open data on Zenodo[2]

6.2 Comparison of heuristic methods

In this section, we compare the RG method with the other heuristics. Since all heuristic methods need one or more initial solutions, first we run RG up to MT, and then we feed the results of RG to all heuristics and run them for additional MT seconds. Note that, since we use a multi-start approach (with k best solutions of RG as starting points) for LS, TS and SA, each neighborhood is explored for $\frac{MT}{k}$ seconds. To guarantee a fair comparison and give the same execution time to all methods, we also run RG for $2MT$.

The results of different scenarios for MT equal to $[60, 600, 1800]s$, averaged among normalized λs between 0 and 1, are presented in Table 1. All the plots in this section are related to the light-constraints scenario, while the results for strict constraints are reported in Appendix. In particular, Figure 5 presents the Cost ratio in a large-scale scenario with varying λ. The *Cost ratio* is computed by Equation (40) considering pure RG running for MT seconds as baseline and each other heuristic – and RG with $2MT$ – as target. As reported, pure RG running for $2MT$ yields the lowest improvement in almost all scenarios. GA moderately improves the initial RG solutions, while LS, TS and SA are the methods that achieve the best results, particularly in large systems, where the improvement is of about 40%.

The same conclusion is drawn when counting how many times a method is the one yielding the lowest cost (i.e., it is the *winner*), as reported in Figure 6a.

Figure 6b reports the number of objective function evaluations for each method across all runs. This depends on different factors, such as the method runtime and the system scale. Indeed, at a larger scale the solution generation process is longer, and a lower number of solutions can be evaluated in a given time limit. Moreover, we only evaluate the feasible solutions, so the number of evaluations also depends on how easy it is to generate a feasible candidate. We can observe that GA and SA have the highest number of objective function evaluations, particularly in small-scale scenarios since the solution generation is faster. On average, among all experiments for light constraints, the improvements of GA, SA, TS, LS methods and RG with $2MT$ over the pure RG are 10.75%, 18.47%, 18.43%, 18.46% and 4.08%, respectively. The same averages for the strict-constraints scenario are 10.16%, 18.84%, 18.57%, 18.81% and 3.78%.

It is worth noting that the worst case in terms of average running time happens when the local and global constraint thresholds are set to slightly larger numbers. Indeed, since each iteration terminates as soon as any constraint is violated, strict thresholds imply more frequent violations and the scenario performs a higher number of iterations in a certain execution time. Since among the best performing heuristics (e.i., LS, TS and SA) LS requires lower number of parameters to fine tune, we chose it for our comparison with state-of-the-art method.

6.3 Comparison with a state-of-the-art method

In this section, we compare the LS algorithm with the *BCPC (Best Cost under Performance Constraint*) method proposed in [7]. A comparison with HyperOpt, another state-of-the-art tool based on Bayesian Optimization, was presented in our previous work [3].

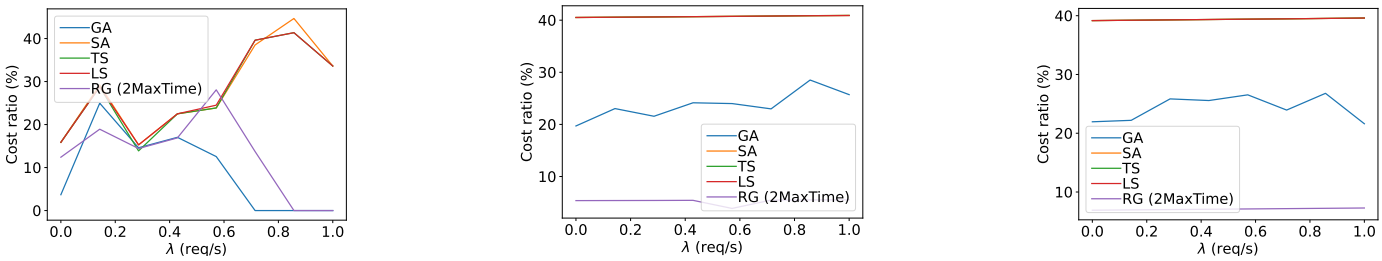
BCPC optimizes DAG-serverless workflows by leveraging a critical path approach to choose the optimal memory configuration for each function, minimizing the execution cost while satisfying a given performance constraint. The authors consider a bundle of candidate memory configurations for each function (component) between 128 MB and 3,008 MB, in 64 MB increments. Their solution considers all possible simple paths between the start and end node of an application DAG and defines the performance constraint such that the sum of the end-to-end response time among all the paths is less than a threshold.

To adapt our problem to [7], we considered a single computational layer of FaaS instances and introduced a global constraint on the whole application execution. To have a fair comparison, we first run BCPC to extract system parameters as the components demand time and cost which relies on AWS Step Functions [40], and then we used them accordingly, to generate 10 different random instances in BCPC with 7, 10 or 15 components as in the previous setting. We considered 46 candidate memory configurations for each function and 100 different performance constraint thresh-

2. https://zenodo.org/record/7436599#.ZGd25uxBz0p , https://zenodo.org/record/7963161#.ZGzch-xBxj8

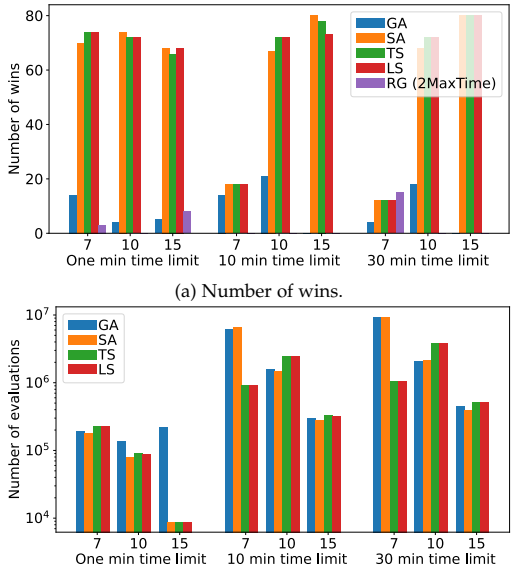
Time limit	#Components	Cost ratio (%) of methods (light constraints) GA	SA	RG ($2MT$)	TS	LS	Cost ratio (%) of methods (strict constraints) GA	SA	RG ($2MT$)	TS	LS
1 min	7	10.35	15.94	4.25	**16.12**	**16.12**	0.68	1.60	0.88	**1.61**	**1.61**
	10	17.49	**28.36**	4.46	28.27	28.27	21.49	32.16	10.10	32.14	**32.18**
	15	9.10	**27.93**	13.06	27.43	27.69	12.97	**30.86**	8.87	30.85	30.25
10 min	7	0.06	**0.09**	0.01	**0.09**	**0.09**	0.03	**0.16**	0.01	**0.16**	**0.16**
	10	5.81	7.28	1.30	**7.30**	**7.30**	9.61	12.55	3.93	12.76	**13.00**
	15	23.71	**40.71**	5.25	40.70	40.70	17.82	**43.10**	3.59	40.09	42.57
30 min	7	0.05	**0.08**	0.04	**0.08**	**0.08**	0.07	**0.15**	0.10	**0.15**	**0.15**
	10	5.94	6.48	1.35	**6.50**	**6.50**	6.19	8.52	1.89	8.81	**8.85**
	15	24.30	**39.38**	7.09	**39.38**	**39.38**	22.64	40.51	4.70	**40.59**	40.51

Table 1: Comparison of the methods, average cost ratio among normalized λ between 0 and 1.



(a) 15 Components, one min time limit. (b) 15 Components, 10 min time limit. (c) 15 Components, 30 min time limit.

Figure 5: Comparison of the proposed heuristic methods, light constraints scenario.



(a) Number of wins.

(b) Number of objective function evaluation.

Figure 6: Heuristic methods number of wins and objective function evaluations, light constraints scenario.

olds, obtained by considering the maximum and minimum memory configuration and normalizing the corresponding end-to-end execution time, considered as global constraint, between 1 and 100 (100 corresponds to the largest execution time threshold, i.e., corresponds to lighter constraint). Running BCPC first, we could also gather the time it needs to determine a solution, which on average ranges in the intervals $[1.17, 2.32]s$, $[7.23, 40]s$ and $[325, 3451]s$, respectively, for scenarios with 7, 10 and 15 components. We set MT to roughly half this time to guarantee a fair comparison.

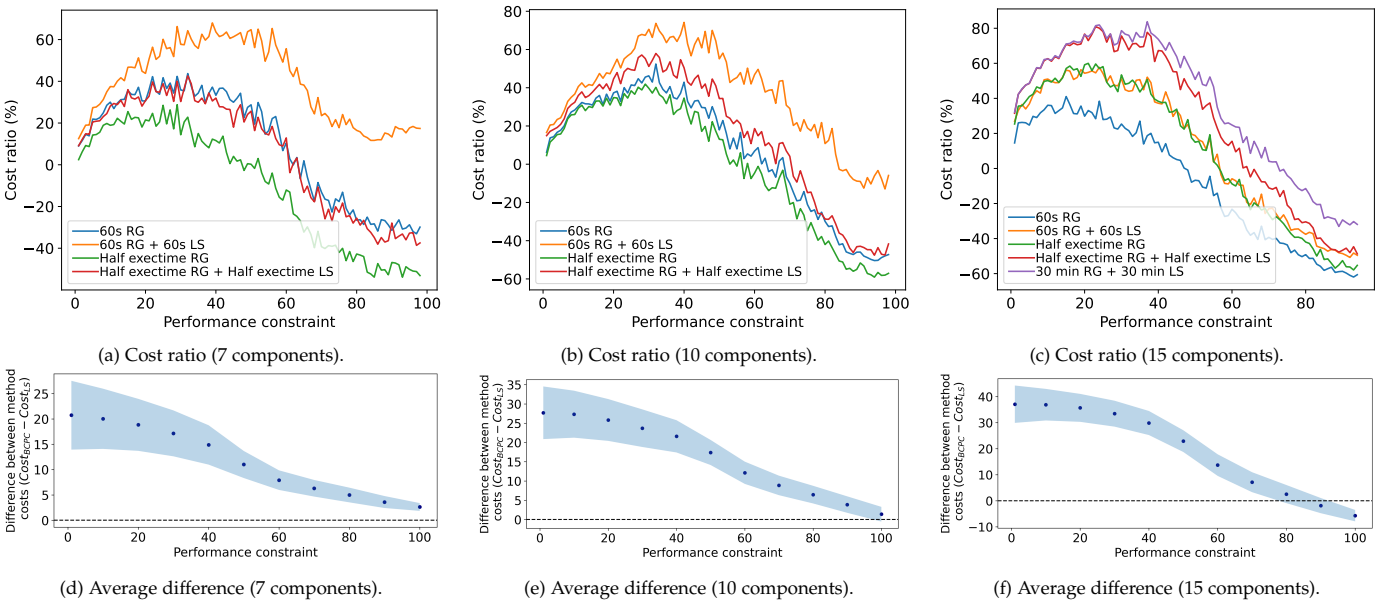
The results are shown in Figures 7a–7c. For the 7-components scenario, if we give equal execution time to the BCPC algorithm and to the RG+LS (red curve in Figure 7a), LS obtains better results than BCPC in 50% of the cases, while if we give two minutes to the RG+LS, the LS always wins and improves the cost ratio up to 70% (orange curve in Figure 7a). A similar comparison for 10 components shows that by running LS for two minutes, it wins most of time with a cost ratio up to 75%, while losses are observed only for few large thresholds (i.e., for light constraints, when the performance constraint is close to 100%), with a cost ratio lower than 10% (see Figure 7b). For the largest scale, LS wins in 63% of the cases with more strict constraints when we run the BCPC algorithm and the RG+LS with the same time limit, and its gain is up to about 81%. Instead, it loses up to 48% in 37% of the cases with light constraints (red curve in Figure 7c). When we raise the running time of RG+LS to one hour, the LS wins in 72% of the cases, and its gain is up to about 84%, while in 28% of the cases LS loses up to 33% (purple curve in Figure 7c). For 15 components, LS gains 27.6% and 36.5% on average, respectively, when the time limit is set to the execution time of the BCPC algorithm or one hour.

We run a *robust ANCOVA* [41] test to assess whether the difference among costs achieved by LS and BCPC, in the presence of a numerical covariate represented by the time constraint threshold, is statistically significant. Figures 7d–7f report the average differences and confidence intervals for the 7, 10 and 15-components scenarios, respectively (where the total execution time of the two approaches is the same). We observe statistical significance (p-value lower than 10^{-4}) in all cases except when the threshold is equal to 100 (in the 10-components scenario) and between 80 and 90 (in the 15-components scenario)

Note that response time is one of the most important constraints for near-real-time AI applications [42]. Consequently, strict thresholds are more critical than light ones for these applications. It can be observed that LS always finds cheaper solutions than the BCPC algorithm under strict and medium-constraints scenarios.

6.4 Real system evaluation

We validated SPACE4AI-D considering a video processing application characterized by seven components, which can be executed locally on edge (computational layers CL_1,



(a) Cost ratio (7 components). (b) Cost ratio (10 components). (c) Cost ratio (15 components).

(d) Average difference (7 components). (e) Average difference (10 components). (f) Average difference (15 components).

Figure 7: Comparison among the solution costs with LS and BCPC [7]. The x-axis reports the normalized global constraint threshold, so that 100 corresponds to the lightest constraint; the y-axis in 7d–7f reports the average difference between method costs, alongside with the confidence intervals: when the confidence intervals do not intersect the horizontal dashed line in $y = 0$, the difference is statistically significant.

CL_2) or remotely in AWS public cloud (CL_3, CL_4, CL_5 and CL_6). The two alternative resources in CL_1 are a Raspberry Pi and an Odroid, which are connected to CL_2 through WiFi. CL_2 includes a private cluster which can host two different VM types, $VM1$ and $VM2$. Each computational layer in cloud has two different VM flavors (m6idn.xlarge and m4.xlarge in CL_3, m4.2xlarge and m4.4xlarge in CL_4, m6id.2xlarge and m5.xlarge in CL_5), and the last computational layer, CL_6, includes two AWS Lambda functions with $8G$ and $9G$ memory size. All computational layers in the cloud are connected together through a fast network (negligible network delay) while CL_2 is connected to the cloud through a 5G with 1 Gbps bandwidth. Further details on the available resources and the components compatibility are provided in Table 6 in Appendix.

In our experiment, 1-minute-long videos are processed in batches of 12 elements. Component C_1 (*ffmpeg_0*) extracts the audio track from the input video, saves it as a file and sends both video and audio to C_2 (*Librosa*). The goal of C_2 is to analyze the audio track to indicate the end of a sentence. Whenever the noise falls below a threshold, C_2 writes the timestamp in a text file and sends it, together with the compressed video, to C_3 (*ffmpeg_1*), which uses the timestamps to cut the video into clips. Component C_4 (*ffmpeg_2*) extracts the audio track and decreases its sample rate to 16000 Hz to make it compatible with C_5 (*DeepSpeech*), which transcribes the audio track of the clips using an open-source recurrent neural network implemented by Mozilla and then sends the transcription text file and the clips to C_6 (*ffmpeg_3*). This extracts from the clips one frame every five seconds, and sends them to C_7 (*object_detector*), which applies a Yolo neural network to detect and label the objects in the frames. The sets of local and global constraints are defined as $\mathcal{LC} = \{\langle C_2, 80s\rangle, \langle C_3, 30s\rangle, \langle C_5, 65s\rangle\}$ and $\mathcal{GC} = \{\langle (C_1, C_2, C_3, C_4, C_5, C_6, C_7), 190s\rangle\}$. The application arrival rate is $\lambda = 25$ req/hour.

The components response times are estimated through ML models trained using the OSCAR-P profiling tool and aMLLibrary as described in [30]. The preliminary evaluation of the performance models accuracy performed in [30] shows a mean absolute percentage error of about 10%. The resources selected by SPACE4AI-D in the optimal solution are reported in Table 6 in the Appendix. The actual application execution time measured in the real system is of about $180s$, which is in line with the $190s$ global constraint threshold, while the value predicted by the ML models is more optimistic, i.e., of $123.6s$. Therefore, the deviation is about 33%. Table 2 reports a comparison between the predicted and actual response times of components involved in local constraints, as well as the cost of application execution. The number of VM instances suggested by SPACE4AI-D for C_2 and C_3 is equal to the one required in the real system to fulfil the local constraints (determined by inspection considering multiple configurations). For C_5, instead, the tool suggested three more VMs, which leads to around 12% prediction error in the total system cost.

	C_2	C_3	C_5
Actual Exec. time	59.33	22.16	59.9
Predicted Exec. time	66.94	17.61	47.64
Exec. time deviation	12%	20%	19%
Actual #VM	2	2	6
Predicted #VM	2	2	9

Actual cost	Predicted cost	Total deviation cost
4.64	5.22	12.4%

Table 2: Predicted and actual time in seconds and cost of the video processing application.

To enhance the robustness of our real system experiment, we defined a baseline method based on exhaustive search, examining all possible placements. This approach is feasible given the relatively small scale of the system and limited search space. Specifically, we have five components that can run on either the Edge or the Cloud, and each component has two alternative candidate resources. Therefore, the search space consists of 32 possible placements. For

each one, we initially set the number of all resources to the maximum available instances and then tentatively reduce the number of edge device and cloud VM instances until the constraints are violated. We observed that our proposed approach found exactly the same solution as the baseline method (see Table 6 in Appendix), which is the globally optimal solution.

7 Related Work

The considerable growth of DL in many application areas during the past few years, on one hand, and the need for powerful resource to run DNNs on the other hand, drew a lot of attention to AI applications component placement in computing continuum, as opposed to traditional data center model. Frequently, optimization is conducted across a spectrum of granularities, spanning from end-devices to edge-cloud resources. For example, [43] developed a Mixed Integer Non-Linear Programming (MINLP) to solve a task allocation problem among end-users' device, fog and cloud. The goal is to minimize the energy consumption of task execution and transmission while guaranteeing delay constraints. They used successive convex approximation and primal-dual decomposition techniques as well as Hungarian algorithm to solve the problem. Similarly, the paper in [44] focuses on minimizing energy consumption in a Mobile Edge Computing (MEC) system by optimizing task offloading and resource allocation. Considering heterogeneous delay constraints and resource competition, a heuristic algorithm incrementally constructs subsets of tasks for offloading and repeatedly solves the offloading sub-problem until further energy reduction is not feasible. Authors in [45] consider applications based on microservices as a DAG and develop a tool to place them at the edge and cloud. The tool clusters the microservices based on their dependencies (data exchanged and number of messages) using Louvain, which is a community detection method, and then deploys the communities that communicate directly with end-users to the edge nodes, and the other communities to the edge or cloud. Similarly, [46] tackled a service-placement problem aiming at minimizing the migration, bandwidth and computation costs while fulfilling the service performance requirements. The authors split the problem into two parts, where the first one demonstrates how to determine the minimum number of CPU resources required for every service to meet the latency requirements, and the second one shows the service placement problem as an integer linear program (ILP). They propose an algorithm leveraging some bounded amount of resource augmentation to solve the ILP. Some recent papers applied Deep Reinforcement Learning (DRL) to solve the resource allocation problem in computing continuum. In [10], the focus was on generating DNN-based tasks from end-devices, considering light-weight and full DNN models. An optimization problem was formulated to minimize delay and error while ensuring task queue stability and inference accuracy. A DRL algorithm made decisions on DNN deployment, task offloading, resource allocation, and channel allocation in each time slot. A sub-optimization problem was addressed using Lyapunov optimization to determine optimal data sizes meeting quality requirements. Differently, in [47], the authors consider multiple inference services, each functioning as an independent component with varying levels of accuracy, to be executed on a computing continuum. They develop a multi-objective optimization problem to maximize the number of deployed services, overall accuracy, and remaining cloud bandwidth, along with a heuristic algorithm for its solution. Likewise, [48] proposes a multi-variant service deployment model for a single independent component on heterogeneous edge devices, focusing on AI applications within the Edge paradigm. This approach is constrained by the limited resources of edge nodes. They also formulate the DNN model variant selection and placement problem, considering inference latency, communication latency, and edge node utilization cost, and propose a heuristic solution.

DNN partitioning is a strategy used to address limited computing resources. Some studies focus on how to place DNN partitions within computing environments. In one such study by [2], experiments were conducted to highlight the benefits of DNN partitioning. They compared running an entire DNN on a mobile device, on a cloud VM, and partially on both. The results indicated that partitioning the DNN reduces latency and energy consumption compared to running it solely on the cloud. Authors in [49] devised a framework to execute DNN inference between the mobile phone and the edge. The framework includes: 1) a *DNN profiler*, which profiles different DNNs from the fine-grained layer and provides performance prediction of DNN layers running on different devices; 2) a *DNN partitioner*, which determines a unique partitioning point by solving a multi-objective optimization problem minimizing the time and energy while guaranteeing a threshold for latency and mobile phone energy consumption; 3) a *Context-Driven Offloading*, which specifies how the inference partitions offload and distribute between the mobile phone and the edge.

To the best of our knowledge, our research addresses a gap in existing literature by accounting for resource contention in application performance estimation, a factor overlooked by previous design-time tools focusing solely on single application instances. The novelty of our paper lays on considering resource contention in the application performance estimation.

8 Conclusions

In this paper, we modeled the design-time component placement and resource selection problem for AI applications in computing continua as an MINLP. We introduced a RG algorithm to address this problem and further refined its solution by incorporating various heuristics including LS, TS, SA, and GA. An extensive experimental analysis shows that all the heuristics always improve the RG solution. Among the best performing heuristics, we compared LS with the BCPC method and we achieved up to 40% cost reduction, 27.6% on average, under the same time limit. The validation in a real prototype environment shows a 12% gap between the actual and predicted costs. As future work, we will consider the runtime problem counterpart, where the input exogenous workload dynamically fluctuates imposing stricter execution time for the problem solution.

Acknowledgment

This work has been funded by the European Commission under the H2020 grant N. 101016577 AI-SPRINT: AI in Secure Privacy pReserving computINg conTinuum.

REFERENCES

[1] D. Schubmehl, *Worldwide Artificial Intelligence Software Platforms Forecast, 2019–2023*, https://www.idc.com/getdoc.jsp?containerId=prUS48881422, 2019.

[2] Y. Kang *et al.*, "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge," in *ACM ASPLOS '17*, 2017.

[3] H. Sedghani *et al.*, "A Random Greedy based Design Time Tool for AI Applications Component Placement and Resource Selection in Computing Continua," in *IEEE EDGE*, 2021.

[4] H. Sedghani *et al.*, "Advancing design and runtime management of ai applications with ai-sprint (position paper)," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2021, pp. 1455–1462.

[5] M. Sponner *et al.*, "Adapting neural networks at runtime: Current trends in at-runtime optimizations for deep learning," *ACM Comput. Surv.*, Apr. 2024.

[6] H. Liang *et al.*, "Dnn surgery: Accelerating dnn inference on the edge through layer partitioning," *IEEE Transactions on Cloud Computing*, vol. 11, no. 3, pp. 3111–3125, 2023.

[7] 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.

[8] U. Tadakamalla *et al.*, "Autonomic resource management for fog computing," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2021.

[9] 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.

[10] 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.

[11] C. Chen *et al.*, "Latency minimization for mobile edge computing networks," *IEEE Transactions on Mobile Computing*, vol. 22, no. 4, pp. 2233–2247, 2023.

[12] F. Filippini *et al.*, "Space4ai-r: A runtime management tool for ai applications component placement and resource scaling in computing continua," in *IEEE/ACM UCC*, 2024.

[13] A. Kambale *et al.*, "Runtime management of artificial intelligence applications for smart eyewears," in *IEEE/ACM UCC*, 2024.

[14] 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.

[15] Q. Li *et al.*, "A latency-optimal task offloading scheme using genetic algorithm for dag applications in edge computing," in *2023 8th ICCCBDA*, 2023, pp. 344–348.

[16] 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.

[17] 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.

[18] D. Ardagna *et al.*, "Adaptive service composition in flexible processes," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 369–384, 2007.

[19] T. Elgamal *et al.*, "Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement," in *IEEE/ACM(SEC))*, 2018.

[20] 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, pp. 2114–2129, 2019.

[21] B. Varghese *et al.*, "Next generation cloud computing: New trends and research directions," *Future Generation Computer Systems*, vol. 79, pp. 849–861, 2018.

[22] M. Shilkov, *Cold Starts in AWS Lambda*, https://mikhail.io/serverless/coldstarts/aws/, 2021.

[23] H. Sivaraman *et al.*, "Task Assignment in a Virtualized GPU Enabled Cloud," in *IEEE HPCS*, 2018.

[24] *Azure Products*, https://azure.microsoft.com/, 2024.

[25] *AWS Products*, https://aws.amazon.com/products/, 2024.

[26] *Welcome to SCAR's documentation*, https://scar.readthedocs.io/en/latest/.

[27] D. M. Naranjo *et al.*, "Accelerated serverless computing based on GPU virtualization," *J. Parallel Distributed Comput.*, vol. 139, pp. 32–42, 2020.

[28] A. Maros *et al.*, "Machine learning for performance prediction of spark cloud applications," in *12th IEEE CLOUD 2019*.

[29] E. Ataie *et al.*, "A Hybrid Machine Learning Approach for Performance Modeling of Cloud-Based Big Data Applications," *The Computer Journal*, pp. 3123–3140, 2021.

[30] E. Galimberti *et al.*, "OSCAR-P and amllibrary: Performance profiling and prediction of computing continua applications," in *ACM/SPEC ICPE 2023*, pp. 139–146.

[31] N. Mahmoudi *et al.*, "Performance modeling of serverless computing platforms," *IEEE Transactions on Cloud Computing*, pp. 2834–2847, 2020.

[32] E. D. Lazowska *et al.*, *Quantitative system performance: computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

[33] M. Kandpal *et al.*, "Role of predictive modeling in cloud services pricing: A survey," in *IEEE Confluence*, 2017.

[34] B. E. Zant *et al.*, "Performance and price analysis for cloud service providers," in *IEEE (SAI)*, 2015.

[35] S. Wang *et al.*, "Online placement of multi-component applications in edge computing environments," *IEEE Access*, vol. 5, pp. 2514–2533, 2017.

[36] E. Aarts *et al.*, *Local Search in Combinatorial Optimization*. Princeton University Press, 2003. [Online]. Available: http://www.jstor.org/stable/j.ctv346t9c (visited on 10/30/2024).

[37] F. W. Glover, "Tabu search - part I," *INFORMS J. Comput.*, vol. 1, no. 3, pp. 190–206, 1989.

[38] J. S. Arora, in *Introduction to Optimum Design (Second Edition)*, Second Edition, San Diego: Academic Press, 2004, pp. 513–530.

[39] J. Manner *et al.*, "Cold Start Influencing Factors in Function as a Service," in *ACM/IEEE UCC Companion*, 2018, pp. 181–188.

[40] *AWS Step Functions Pricing*, https://aws.amazon.com/step-functions/pricing/, 2024.

[41] P. Mair *et al.*, "Robust statistical methods in R using the WRS2 package," *Behavior Research Methods*, vol. 52, pp. 464–488, 2020.

[42] A. Aslam *et al.*, "Investigating response time and accuracy in online classifier learning for multimedia publish-subscribe systems," *Multimed Tools Appl*, vol. 80, pp. 13 021–13 057, 2021.

[43] B. Kopras *et al.*, "Task allocation for energy optimization in fog computing networks with latency constraints," *IEEE Transactions on Communications*, vol. 70, no. 12, pp. 8229–8243, 2022.

[44] J. Mei *et al.*, "Energy-efficient heuristic computation offloading with delay constraints in mobile edge computing," *IEEE Transactions on Services Computing*, vol. 16, no. 6, pp. 4404–4417, 2023.

[45] A. Rahmanian *et al.*, "Microsplit: Efficient splitting of microservices on edge clouds," in *IEEE/ACM(SEC)*, 2022, pp. 252–264.

[46] I. Cohen *et al.*, "Dynamic service provisioning in the edge-cloud continuum with bounded resources," *IEEE/ACM Transactions on Networking*, pp. 1–16, 2023.

[47] L. Wang *et al.*, "Mpdm: A multi-paradigm deployment model for large-scale edge-cloud intelligence," *IEEE Internet of Things Journal*, vol. 10, no. 10, pp. 8773–8785, 2023.

[48] M. Bensalem *et al.*, "Dnn placement and inference in edge computing," in *43rd MIPRO*, 2020, pp. 479–484.

[49] Y. Huang *et al.*, "AoDNN: An Auto-Offloading Approach to Optimize Deep Inference for Fostering Mobile Web," in *IEEE INFOCOM*, 2022.

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 and optimization for resource management in computing continuum.

Federica Filippini received her M.Sc. degree in Mathematical Engineering and her Ph.D. degree in Information Technology at Politecnico di Milano, where she is now a post-doc researcher. Her research interests include optimization problems applied to resource selection and scheduling in Cloud and distributed environments.

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

For the sake of space, we provide the tables summarizing the parameters and variables of our MINLP formulation, the description of the crossover operator of the Genetic Algorithm and the experimental results related to the strict-constraints scenario in the following subsections.

Parameters and decision variables

The parameters and decision variables of our problem formulation are shown in Table 3 and Table 4, respectively.

Parameters - components	
$\mathcal{I}$	Set of component indices
λ_i	Incoming workload of component i
p^{ik}	Probability of running component k after component i
δ_{ik}	Amount of data transferred between components i and k
$\mathcal{C}^i$	Set of candidate deployments c_s^i for component i
$\mathcal{H}_s^i$	Set of indices h such that partition π_h^i is included in c_s^i
$\tilde{\lambda}_h^i$	Incoming workload of partition π_h^i
$\tilde{p}_{h\xi}^i$	Probability of running partition π_ξ^i after partition π_h^i
$\tilde{\delta}_{h\xi}^i$	Amount of data transferred between partition π_h^i and partition π_ξ^i
$\tilde{m}_h^i$	Memory requirement of partition π_h^i
Parameters - resources	
$\mathcal{J}_\mathcal{C}$	Set of indices of all VM types in the cloud backend
$\mathcal{J}_\mathcal{E}$	Set of indices of all devices available on the edge
$\mathcal{J}_\mathcal{F}$	Set of indices of all possible function configurations
$\mathcal{J}$	Set of indices of all computing continuum resources
$\mathcal{L}^l$	Set of resource indices at layer l
n_j	Number of available devices of type $j \in \mathcal{J}_\mathcal{C}$
a_{hj}^i	1 if partition π_h^i can be executed on device $j \in \mathcal{J}$
c_j^E	Execution cost on device type $j \in \mathcal{J}_\mathcal{E}$
c_j^C	Execution cost on VM type $j \in \mathcal{J}_\mathcal{C}$
$c_{hj}^{i,F}$	Execution cost of partition π_h^i on function $j \in \mathcal{J}_\mathcal{F}$
c^T	Transition cost for AWS lambda functions
D_{hj}^{il}	Demanding time of partition π_h^i on device $j \in \mathcal{L}^l \subseteq (\mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C})$
d_{hj}^i	Average execution time of partition π_h^i on $j \in \mathcal{J}_\mathcal{F}$
$d_{hj}^{i,hot}$	Hot request execution time of partition π_h^i on $j \in \mathcal{J}_\mathcal{F}$
$d_{hj}^{i,cold}$	Cold request execution time of partition π_h^i on $j \in \mathcal{J}_\mathcal{F}$
Parameters - network	
$\mathcal{D}$	Set of existing network domains
$\mathcal{ND}^d$	Set of layers l included in the network domain d
a^d	Access time of network domain d
B^d	Bandwidth of network domain d
$\mathcal{UL}^d$	Set of indices of all devices contained in any $l \in \mathcal{ND}^d$
Parameters - constraints	
$\mathcal{LC}$	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
$\overline{LR}_i$	Upper bound threshold for the response time of component i
$\overline{GR}_P$	Upper bound threshold for the total response time of a path P
Other parameters	
λ	Input exogenous workload
T	Time unit (one hour)
M_E	Constant for equilibrium conditions in edge
M_C	Constant for equilibrium conditions in cloud
M_N	Constant for defining network delays

Table 3: Problem Parameters

Decision Variables	
x_j	1 if device $j \in \mathcal{J}$ is used, 0 otherwise
z_s^i	1 if deployment c_s^i is selected for component i, 0 otherwise
y_{hj}^i	1 if partition π_h^i runs on device $j \in \mathcal{J}$, 0 otherwise
$\widehat{y}_{hj}^i$	Number of instances of device $j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$ assigned to partition π_h^i
$\bar{y}_j$	Maximum number of running devices of type $j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$
$\tilde{w}_{h\xi}^{id}$	1 if partitions π_h^i and π_ξ^i are consecutive and deployed on different devices in the same network domain d
$\hat{w}_{h\xi}^i$	Index of the network domain exploited by π_h^i and π_ξ^i to communicate
$f_{h\xi}$	1 if π_h^i and π_ξ^k are executed in different resources
$\tilde{t}_{h\xi}^i$	Network transfer time between π_h^i and π_ξ^k
t^i	Intra-component expected delay
$w_{h\xi}^{ik}$	Index of the network domain exploited by π_h^i and π_ξ^k to communicate
$\tilde{t}_{h\xi}^{ik}$	Network transfer time between π_h^i and the first partition of component k
α_h^i	1 if π_h^i is the first partition of component i, 0 otherwise
$\tilde{R}_h^i$	Response time of partition π_h^i
U_j	Utilization of device $j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$
R^i	Response time of component i
$\hat{R}_P$	Response time of a path P
C_E	Cost of edge devices
C_C	Cost of cloud Virtual Machines
C_F, C_T	Execution cost and transition cost of FaaS configurations

Table 4: Decision Variables

Algorithm 2 MixParts

Input: $Part1$, $Part2$
$Children \leftarrow \emptyset$
$Child1 \leftarrow Part1 + Part2$
$ActiveCL1, ActiveCL2 \leftarrow$ Get all active CLs in $Part1$ and $Part2$
$SharedCL \leftarrow ActiveCL1 \cap ActiveCL2$
if $SharedCL$ is not None **then**
  $TwoChildren \leftarrow False$
  for $cl \in SharedCL$ **do**
    $ActiveRes1 \leftarrow$ Get active resource of cl in $Part1$
    $ActiveRes2 \leftarrow$ Get active resource of cl in $Part2$
    if $ActiveRes1 == ActiveRes2$ **then**
      Continue
    else
      if Not $TwoChildren$ **then**
        $Child2 \leftarrow Part1 + Part2$
        $TwoChildren \leftarrow True$
      end if
      Move all partitions running on $ActiveRes2$ to $ActiveRes1$ and $ActiveRes1$ to $ActiveRes2$ in $Child1$ and $Child2$, respectively.
    end if
  end for
end if
$Children \leftarrow Children \cup Child1$
if $TwoChildren$ **then**
  $Children \leftarrow Children \cup Child2$
end if
return $Children$

Genetic Algorithm Crossover Operator Implementation

As discussed in Section 5.2.4, due to some constraints specific to our problem, such as selecting only one resource in each computational layer, mixing directly the chromosome parts will likely result in an infeasible solution. To avoid this and make the search more effective, in the implementation of the crossover operator we mix the two parents chromosome parts preserving the feasibility of the assignment by relying on Algorithm 2. We first mix the two chromosome parts to generate $Child1$ and then, for both parts, we extract $ActiveCLList$ (see Section 5.2.1) and get

(a) Light scenario.

Scenario	#Components	#Nodes in Computational Layers (CL)									#Local and global constraints
		CL_1	CL_2	CL_3	CL_4	CL_5	CL_6	CL_7	CL_8	CL_9	
1	7	Edge: 2	Edge: 3	Edge: 3	VM: 4	VM: 4	FaaS: 2	-	-	-	3, 3
2	10	Edge: 2	Edge: 3	Edge: 3	VM: 4	VM: 4	FaaS: 3	-	-	-	4, 4
3	15	Edge: 2	Edge: 4	Edge: 4	Edge: 4	VM: 5	VM: 5	VM: 5	VM: 5	FaaS: 5	5, 5

(b) Strict scenario.

Scenario	#Components	#Nodes in Computational Layers (CL)							#Local and global constraints
		CL_1	CL_2	CL_3	CL_4	CL_5	CL_6	CL_7	
1	7	Edge: 1	Edge: 3	VM: 4	VM: 4	FaaS: 3	-	-	3, 3
2	10	Edge: 1	Edge: 4	Edge: 4	VM: 4	VM: 4	FaaS: 4	-	4, 4
3	15	Edge: 1	Edge: 4	Edge: 5	VM: 5	VM: 5	VM: 5	FaaS: 5	5, 5

Table 5: Comparative analyses parameters.

Resources and components	#Candidate Nodes in Computational Layers (CL)					
	CL_1	CL_2	CL_3	CL_4	CL_5	CL_6
Candidate resource name	(RasPi, Odroid)	(VM1, VM2)	(m6idn.xlarge, m4.xlarge)	(m4.2xlarge, m4.4xlarge)	(m6id.2xlarge, m5.xlarge)	(AWS_λ_1, AWS_λ_2)
Resource cost ($/h)	(0.6, 0.8)	(0.675, 0.85)	(0.318, 0.2)	(0.4, 0.8)	(0.4746, 0.192)	(0.56196, 0.6325)
Number of cores	(4, 4)	(4, 4)	(4, 4)	(8, 16)	(4, 8)	(-, -)
Number of instances	(12, 6)	(8, 4)	(12, 6)	(6, 3)	(12, 6)	(1, 1)
Components	(C_1)	(C_2)	(C_3)	(C_4)	(C_5)	(C_6, C_7)
Selected resources	RasPi	VM1	m4.xlarge	m4.2xlarge	m5.xlarge	(AWS_λ_1, AWS_λ_2)

Table 6: Real system configuration.

the shared active layers between the two chromosome parts (lines 1-5). If there are no shared active layers, it means that the chromosomes can be mixed directly without any violations; otherwise, we must change the generated child to avoid violations. Therefore, for each shared active layer, we get the resources used in the two chromosome parts as active resources (lines 6-10). If both active resources are the same, no changes are needed (lines 11-12). Otherwise, we generate a second solution $Child2$ so that, at the current layer, the first child selects the active resource coming from the first chromosome part, and the second child selects the one coming from the second gene part (lines 13-21). Finally, all the generated children are returned (lines 22-26).

Experimental setup parameters

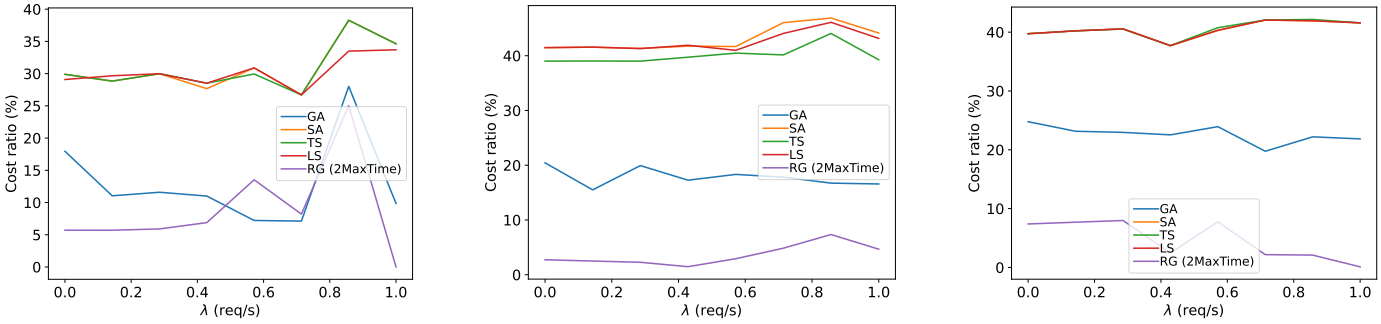
The details of experimental setup related to light and strict scenarios is presented in Table 5 and the details of real system configuration is provided in Table 6.

Strict constraints results

This section summarizes the results of the same analyses reported in Section 6.2 related to strict constraints where the local and global constraints are about 10 times lower than light constraints and λ is considered 10 times higher than the light version. As in the previous sections, λ is normalized between 0 and 1.

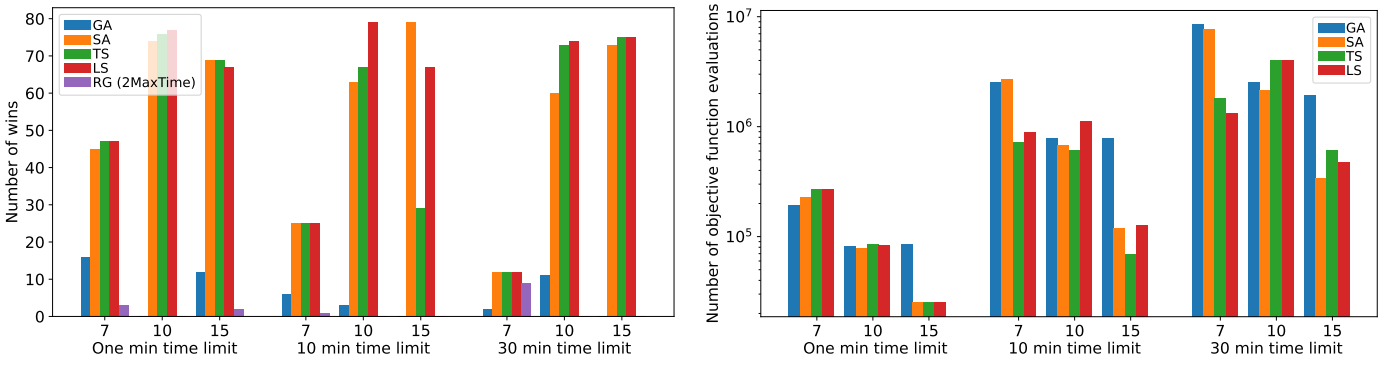
Comparison of heuristic methods

Figure 8 shows the comparison among all the heuristic methods. Similar to light constraints scenario, LS, TS and SA obtain similar and better results compared with the RG and GA. Figure 9a confirms that these three heuristic methods are the winners in most of the cases and Figure 9b shows that GA makes the largest number of objective function evaluations being more efficient in generating random solutions.



(a) 15 Components, one min time limit. (b) 15 Components, 10 min time limit. (c) 15 Components, 30 min time limit.

Figure 8: Comparison of the proposed methods, strict constraints scenario.



(a) Number of wins. (b) Number of objective function evaluation.

Figure 9: Heuristic methods number of wins and objective function evaluations, light constraints scenario.