A Random Greedy based Design Time Tool for AI Applications Component Placement and Resource Selection in Computing Continua

Hamta Sedghani, Federica Filippini, Danilo Ardagna

April, 22, 2021

Abstract

Artificial Intelligence (AI) and Deep Learning (DL) are pervasive today, with applications spanning from personal assistants to healthcare. Nowadays, the accelerated migration towards mobile computing and Internet of Things, where a huge amount of data is generated by widespread end devices, is determining the rise of the edge computing paradigm, where computing resources are distributed among devices with highly heterogeneous capacities. In this fragmented scenario, efficient component placement and resource allocation algorithms are crucial to orchestrate at best the computing continuum resources. In this paper, we propose a tool to effectively address the component placement problem for AI applications at design time. Through a randomized greedy algorithm, our approach identifies the placement of minimum cost providing performance guarantees across heterogeneous resources including edge devices, cloud GPU-based Virtual Machines and Function as a Service solutions. Finally, we compare the random greedy method with the *HyperOpt* framework and demonstrate that our proposed approach converges to a near-optimal solution much faster, especially in large scale systems.

Keywords: Component placement, resource selection, AI applications.

1 Introduction

In the last few years, the cloud computing paradigm led to significant growth of Artificial Intelligence (AI) and Deep Learning (DL) pervasiveness: the total worldwide spending on cloud services is expected to surpass 1.3 trillion in revenue by 2025, at a CAGR of 16.9% [1]. The main strength of this paradigm is given by the fact that it makes an ideally unlimited computational and storage power accessible according to pay-to-go pricing models [2]. However, cloud resources are far from end users and Internet of Things (IoT) devices where data are generated, and such distance can cause long delays. Exploiting resources at the edge of the network can reduce latency, save bandwidth and increase energy

efficiency and privacy protection. Accordingly, edge computing was proposed as an alternative to cloud and its adoption is steadily increasing [3]. Such computing paradigm exploits devices with highly heterogeneous capacities. Therefore, component placement and resource allocation algorithms are crucial to orchestrate at best the physical resources of the computing continuum, minimizing the expected execution costs while meeting DL model accuracy, application performance, security and privacy constraints.

The primary goal of these algorithms is to determine, at design time, the optimal deployment of all AI application components, characterized by heterogeneous requirements in terms of, e.g., computational and storage power, on the candidate resources available in the computing continuum. Such deployment may then be adapted at runtime to deal with workload fluctuations causing resources saturation or under-utilization. Moreover, AI applications frequently include Deep Neural Network (DNN) components that might be partitioned differently according to resources capacity and network settings. Each partition can run on a device independently. Accordingly, in the following we will use *partition placement* and *component placement* alternatively.

This paper proposes a design-time tool to tackle the component placement problem and resource selection in computing continua, effectively addressing resource contention by adopting queueing theory to model application components response times. We developed an efficient randomized greedy algorithm, that identifies the hardware to buy on the edge and the minimum-cost placement across heterogeneous resources including edge devices, cloud GPU-based Virtual Machines and Function as a Service solutions, under Quality of Service (QoS) response time constraints.

To the best of our knowledge, this is the first work that considers AI applications with different candidate deployments which include different DNN partitions for each component and different resource candidates and aims to select the optimal deployment and resource candidate while taking into account resource contention. Through an extensive experimental campaign, we highlight that our approach can yield a significant improvement compared with the *HyperOpt* framework [4], based on Bayesian methods, by decreasing the average running time by a factor of 120–160.

The remainder of this paper is organized as follows. Section 3 introduces the application and the computing continuum model considered in this work. Section 4 describes the use case we address in the paper, providing an example of the benefits our result would produce in real-life scenarios. Section 5 presents the problem statement and the algorithm we propose to tackle it. Experimental results are discussed in Section 6, while Section 2 describes the related works. Conclusions are finally drawn in Section 7.

2 Related Work

Components placement in computing continua has recently received a lot of attention from the research community. Authors of [5] recently proposed a clas-

sification of the literature proposals according to the layers involved (cloud/fog-edge nodes/end devices), the purpose of the placement (e.g., end devices offload, fog nodes offload, fog nodes cooperation, data distribution among fog nodes or cloud, etc.), the decisions taken (e.g., tasks priority, resource to task assignment, hardware resources placement, etc.), the relevant metrics (latency, energy, profit-cost, and device-specific), and the general goal (e.g., energy minimization, latency-throughput trade-off, privacy and security of data).

As a recent paper close to our work, [6] proposed a framework to provide a deadline-aware service placement on fog/cloud. The service placement is performed according to different application requirements, such as memory, processing power, bandwidth, and application deadlines. The framework extracts the applications modules and, for each module, it obtains a detailed description of the requirements. Then, it applies a Genetic Algorithm to find the best placement. Each gene in a chromosome denotes a module, and its value represents a resource assigned to it.The fitness function is the residual time to the deadline of the application. The authors used iFogSim simulator to validate the proposed approach in terms of fulfilment of the service deadlines.

Authors in [7] propose Pogonip, an edge-aware scheduler integrated with Kubernetes, that is designed for asynchronous microservice-based applications. They model the placement of microservices in edge-cloud environments as an Integer Linear Programming problem, which minimizes operational costs under network latency constraints, and develops a greedy algorithm to determine a solution both for the edge-placement and the cloud-placement subproblem.

Deep Neural Networks (DNNs) partitioning is a commonly adopted approach to reduce model size and computational requirements. While traditional approaches partition the DNNs by layers, authors in [8] propose a novel framework, where DNN models are split horizontally, so that each partition (called sub-task) has a smaller version of all layers. In such context, they propose a reinforcement learning-based algorithm for resource allocation, creating an inference dependencies table shared across the whole system.

Some works focus on Mobile Edge Clouds, with a particular emphasis on the network resources. For example, [9] proposes an Integer Linear Programming problem and a greedy heuristic algorithm to tackle the cloudlet placement problem in a Wireless Metropolitan Area Network, minimizing the average delay between mobile users and the cloudlets. In MECs, user mobility is also particularly relevant. [10] models the service infrastructure placement problem in a fog computing environment as a latency and capacity-constrained location selection problem. The placement problem in a Fog Computing/NFV environment is modeled as a Mixed-Integer Linear Programming problem, and tackled through a Simulated Annealing algorithm coupled with a Tabu Search, considering 5G mobile network requirements.

Among the proposals, [11], [12] and [13] are the closest to our approach. Authors in [11] formulate an offline version of a multi-component application placement problem as a Mixed Integer Linear Program, solved by the CPLEX solver. The solution of the offline problem is used as a lower bound to estimate the performance of an online algorithm based on simple heuristic techniques

such as iterative matching and local search. [12] investigates the placement of multi-component applications in the edge. Application components are modeled as an application graph while physical edge devices as a physical graph. Both online and offline algorithms are proposed to optimally map the application to the physical graph while providing performance guarantees for the end applications. Finally, [13] tackles the problem of determining which tasks should be deployed on edge or cloud resources, in the context of the FaaS paradigm. It proposes a dynamic task placement framework to minimize latency subject to cost constraints or to minimize costs subject to latency constraints.

The novelty of our paper lays on the fact that the design time tools proposed so far in the literature, to the best of our knowledge, consider only a single application instance running on the available resources, so that resource contention is never considered in the estimate of application performance as the DN network optimal partition point selection.

3 Application and Resource Models

In this section, we provide an overview of the general model developed for the design-time component placement and resource selection problem tackled in our work. In particular, we discuss the application components model and the Quality of Service (QoS) requirements in Section 3.1, while we describe the computing continuum resources model, the network model and the system costs in Sections 3.2 and 3.3, respectively.

3.1 Application components model and QoS requirements

In our framework, AI applications are modeled as directed acyclic graphs (DAGs), see Figure 1, whose nodes represent the different components. These are Deep Neural Networks (DNNs) implemented as Python functions running in Docker containers that can be deployed in edge devices, cloud Virtual Machines (VMs) or according to the Function as a Service (FaaS) paradigm. For the sake of simplicity [14, 15, 16], 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. We denote the set of components by $\mathcal{I}$. The directed edge connecting components i and $k \in \mathcal{I}$ is labelled with $\langle p^{ik}, \delta^{ik} \rangle$, where p^{ik} is a transition probability, and δ^{ik} denotes the size of data sent from i to k. Furthermore, components can be characterized by multiple candidate deployments. Each element in a candidate deployment is a partition of the corresponding DNN. 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 2. 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 $\mathcal{H}_3^1 = \{5, 6, 7\}$. Note that, in some cases, one of the candidate deployments may correspond to the complete DNN identifying the component. The corresponding set c_s^i would therefore contain a unique partition π_h^i.

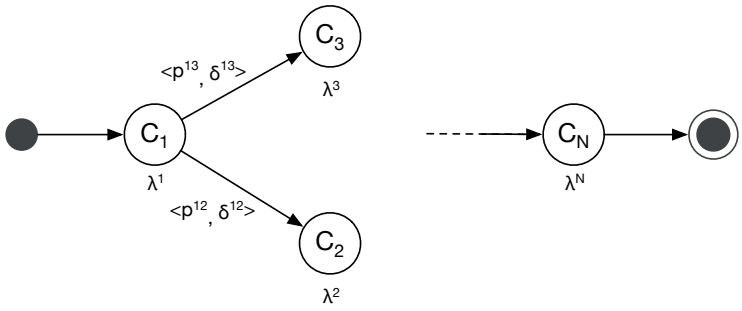
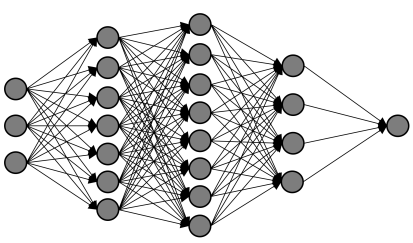
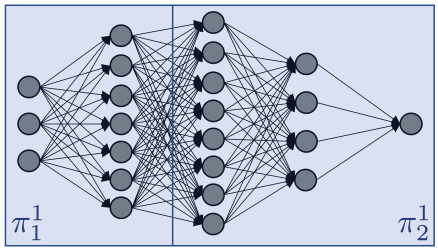


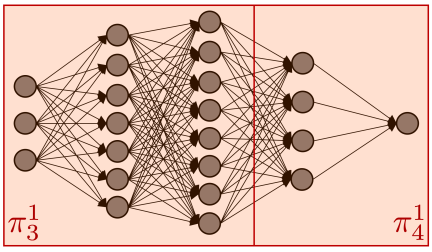
Figure 1: Directed acyclic graph for components.



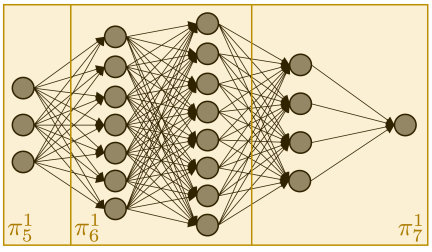
(a) Component 1



(b) Candidate deployment c_1^1



(c) Candidate deployment c_2^1



(d) Candidate deployment c_3^1

Figure 2: Example of AI application component with its candidate deployments

Together with p^{ik}, which is related to the transition from component i to component k, we introduce an additional parameter $\tilde{p}_{h\xi}^i$, which defines the probability that partition π_ξ^i is executed just after partition π_h^i. Mechanisms as early stopping [17] entail that not all component partitions are necessarily executed,

which dictates the necessity of defining the probability of actually moving from one to the other. Similarly, we define $\tilde{\delta}^i_{h\xi}$ as the amount of data transferred from partition π^i_h to partition π^i_ξ. Moreover, each π^i_h is characterized by a memory requirement (expressed in MB), and by a total load λ^i_h, which depends on λ and on the transition probabilities related to all predecessors of the partition.

For simplicity, we consider DAGs including only sequential execution and branches, since, as in [14], we assume that loops are unfolded (or peeled) while parallel execution is not supported for the time being. We define execution paths as sequences of application components from the entry point to the exit point of the DAG, while a path P denotes a set of consecutive components included in an execution path.

The main performance metric we consider in our system is the response time. QoS requirements may be imposed on both the response time of single components (*local constraints*), and on the response time of all components included in a path (*global constraints*).

3.2 Resources general model

Computing continuum resources include edge devices, cloud Virtual Machines (VMs) and Function as a Service (FaaS) configurations. Each resource is characterized by a maximum memory capacity and is included in a different computational layer. In our model, the first layer includes local devices generating data (such as drones, see Section 4). The second layer is often located in the edge and may include smartphones, PC or edge servers with a higher computational power. Cloud layers include VMs coming from a single cloud provider catalogue (however, our approach can be easily extended to consider multiple cloud providers). The VMs selected at a given layer are homogeneous and evenly share the workload due to the execution of one or multiple application components. We consider VMs characterized by a single GPU, if available: costs and inference performance scale linearly with the number of GPUs [18], therefore such assumption allows to improve the availability of the whole system. Finally, we consider all FaaS configurations to be in the same layer because functions run on independent containers. The same component can be associated with multiple FaaS configurations characterized by different memory settings.

The response time of all the executed components is computed as follows:

- We characterize the demanding time to run a component on edge or cloud resources without resource contention (i.e., when a node executes a single request, see [19]).
- We model edge devices and VM instances as individual M/G/1 (single server multiple class) queues [20] to cope with resource contention.
- We compute the average execution time for each component on a given FaaS configuration starting from the execution times of hot and cold requests, the expiration threshold and the arrival rate of the configuration by relying on the tool proposed in [21].

- We consider several network domains connecting edge devices with each other and with the remote cloud back-end. Resource layers are included in, possibly, multiple network domains, associated with a given technology characterized by access time and bandwidth.

- We include in the global execution time of each path the network delay due to data transmissions, depending (see, e.g., [20]) on the amount of data transferred, the network bandwidth and the access delay of the network domain. We neglect the network delay in the cloud since all VMs and FaaS instances are executed in the same data center.

According to the results reported in [20] and [21], response times of components deployed at each layer can be estimated with a percentage error between 10% and 30%, which is acceptable for design-time purposes [19].

Finally, a compatibility matrix $\mathbf{A}$ is introduced to show which devices can be used to execute each partition. Specifically, a^i_{hj} is 1 if π^i_h can be executed on device j, 0 otherwise.

3.3 System costs

The costs related to the computing continuum resources are:

- Edge devices costs are estimated, for the single run of the target application, amortizing the investment cost along the lifetime horizon of the device and dividing the yearly management costs by the number of times the application is run over a year.

- Cloud VMs costs are hourly costs [22], while FaaS costs are expressed in GB-second [23], and they depend on the memory size, the functions duration, and the total number of invocations.

- An additional *transition cost* can be required by FaaS providers (e.g., [24]) to account for the message passing and coordination. Some third party frameworks, e.g., [25], avoid transition costs by supporting the orchestration through an architectural component.

In the next section, we introduce a reference use case that will be quantitatively analysed deeply in Section 6.

4 A running example

To motivate our work and the obtained results, we investigate a use case related to the maintenance and inspection of a wind farm. The identification of damages in wind turbines blades is performed in the computing continuum, based on images collected by drones. The application software is characterized by multiple components, consisting of DNNs that can be deployed and executed

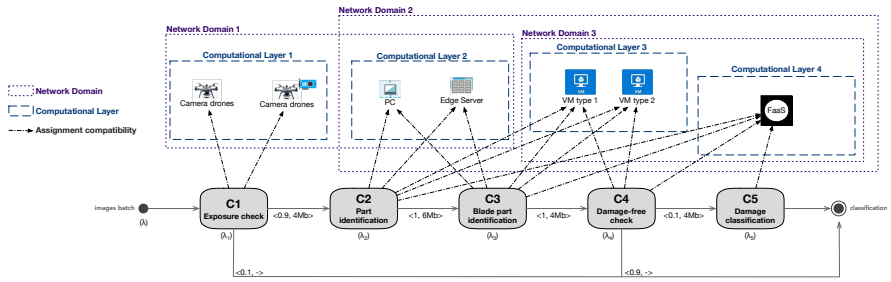


Figure 3: A Use case of identifying wind turbines blade damage.

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 paradigm). The set of components is illustrated in Figure 3. They can be deployed overall on four layers, and 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 $\mathbf{A}$).

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 our tool), controlled remotely by a human operator, takes pictures of the wind turbine. These are composed of three blades, and pictures must be collected, for each blade, from four different angles, to account for different types of damages; therefore, a huge amount of data is collected.

Images are processed in batches which define the incoming workload λ. Each batch is subject to an *exposure check* (C_1), which determines if the image quality is sufficient for further processing. If not, the component triggers the acquisition of new images. This improves the efficiency of the whole inspection process, since it allows to immediately react to the need for further data acquisition.

All well-exposed pictures are inspected by a sequence of two components (C_2 and C_3) which collectively implement a complex, computer vision-based application whose goal is to monitor the inspection campaign and guarantee that this covers the complete site. They take as inputs the images processed by C_1 and a model of the wind farm, and, positioning the pictures on the farm itself, guide the operator to identify the next element to be examined. In particular, C_2 is responsible for the preliminary analysis of the images like identifying if it is a part of the blade or not, while C_3 identifies the parts and the position of the image elements on the blade. These components, especially C_3, require a significant amount of computing and storage power. However, executing them at the edge may help in reducing the time needed to complete the maintenance and inspection process, if further data is needed to better identify the damages.

Images are then processed by two additional AI modules. C_4 is responsible for a *damage-free check*, i.e., for assessing whether the inspected part is damaged or not (this may possibly require the acquisition of new images). Depending on the situation, it may happen that a high percentage of the acquired pictures

is clear, namely it does not show a damage. Finally, C_5 is responsible for classifying the damage. These last steps are characterized by heavy computation requirements, therefore they are always performed in the cloud. Cloud resources are based on VMs and on the FaaS paradigm. FaaS includes different function configurations with different memory allocated and only one component can be run on each container with the specified function configuration.

The four computational layers, namely the one involving camera drones, the one of edge resources, and those including Virtual Machines and FaaS, respectively, belong to three different network domains. In particular, drones and all edge resources communicate through a Wi-Fi network. Virtual Machines and the FaaS configurations are connected via a fiber optic network, while data is transferred from edge to cloud resources through a 4G or 5G network. For what concerns the candidate deployments, C_1 has a single, one-partition deployment, C_2, C_4 and C_5 have two deployments with one and two partitions while C_3 has three deployments with one, two and three partitions. The transition probabilities p^{ik} and the amount of data δ^{ik} transferred between components are reported in Figure 3.

This scenario is characterized by both local and global QoS constraints, as described in Section 3.1. In particular, we prescribe that component C_5 must have a maximum response time of 2.5s, while we enforce that the global response time of the first four components does not exceed 2s.

5 Problem Statement and Solution

This section provides an overview of how we modeled the applications component placement and resource selection problem on heterogeneous edge and cloud resources. We developed a Mixed Integer Non-Linear Programming (MINLP) optimization formulation, aiming at minimizing the deployment cost at design time, while satisfying local and global QoS requirements. For space limits, we focus here only on the objective function of our model, while the complete formulation is available as a technical report [26].

The main goal of our tool is to determine which kind of resource we should select at each computational layer (that includes which hardware to buy on the edge or, e.g., which type/flavor of VM to use in the cloud), whether each component partition should be deployed on the given resource, and, in this case, if the assignment is compatible with: 1) memory constraints, used to determine the maximum number of components partitions that can be co-located in each device, and 2) QoS requirements.

We denote by $\mathcal{J}$ the set of all resources in the computing continuum. To define the assignment decisions, namely to characterize which resources we should select at each computational layer and how components are assigned to the available devices, we introduce the following variables:

- y^i_{hj}, which, for all $i \in \mathcal{I}$, for all $c^i_s \in \mathcal{C}^i$ and for each $h \in \mathcal{H}^i_s$ and $j \in \mathcal{J}$, is equal to 1 if partition π^i_h is deployed on device j,

- x_j, which is 1 if device $j \in \mathcal{J}$ is used in the final deployment,
- $\widehat{y}^i_{hj}$, which, for cloud resources, denotes the number of VMs of type j assigned to any partition π^i_h.

As introduced in Section 3.3, 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. In order to compute them, we denote with T the overall time an application is active for a single run and we assume that T is equal or less than one hour.

If we denote by $\mathcal{J}_\mathcal{E}$ the subset of $\mathcal{J}$ storing all the available edge devices, the corresponding execution cost can be defined by $C_E = \sum_{j \in \mathcal{J}_\mathcal{E}} c^E_j x_j$, where c^E_j is the amortized cost of the edge device $j \in \mathcal{J}_\mathcal{E}$. The total execution cost on cloud VMs, whose set will be denoted by $\mathcal{J}_\mathcal{C} \subseteq \mathcal{J}$, can instead be computed as $C_C = \sum_{j \in \mathcal{J}_\mathcal{C}} c^C_j \bar{y}_j$, where c^C_j and $\bar{y}_j = \max_{i,h} \widehat{y}^i_{hj}$ denote the hourly cost and the maximum number of running VMs of type $j \in \mathcal{J}_\mathcal{C}$, respectively.

Finally, let $\mathcal{J}_\mathcal{F}$ denote the set of all FaaS configurations, and let $c^{F,i}_{hj}$ be the GB-second unit cost for executing π^i_h on the function configuration $j \in \mathcal{J}_\mathcal{F}$. FaaS total costs depend on the memory size, the functions duration, and the total number of invocations. The execution cost of the function layer will be as follows:

$$C_F = \sum_{i \in \mathcal{I}} \sum_{s : c^i_s \in \mathcal{C}^i} \sum_{h \in \mathcal{H}^i_s} \sum_{j \in \mathcal{J}_\mathcal{F}} c^{F,i}_{hj} d^{i,hot}_{hj} y^i_{hj} \tilde{\lambda}^i_h T, \tag{1}$$

where $d^{i,hot}_{hj}$ denotes the execution time of a hot request of partition π^i_h on function configuration $j \in \mathcal{J}_\mathcal{F}$. Note that the cost of the used memory is embedded in $c^{F,i}_{hj}$. Indeed, $d^{i,hot}_{hj}$ is inversely proportional to the memory $\tilde{m}^i_h$ allocated to the partition π^i_h (see [27]). According to some FaaS providers (see, e.g., *AWS Step Functions* [24] and *Azure Logic Apps* [28]), 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., *SCAR* and *OSCAR* [25]), the state transition cost is set to $c^T = 0$. Without loss of generality, we can thus formulate the transition cost as:

$$C_T = \sum_{i \in \mathcal{I}} \sum_{s : c^i_s \in \mathcal{C}^i} \sum_{h \in \mathcal{H}^i_s} \sum_{j \in \mathcal{J}_\mathcal{F}} c^T y^i_{hj} \tilde{\lambda}^i_h T. \tag{2}$$

Therefore, the objective function of our problem, which corresponds to the minimization of all the operational costs defined in the previous equations, reads:

$$\min C_E + C_C + C_F + C_T \tag{3}$$

subject to assignment compatibility, memory and QoS constraints and to the selection of a single device at each layer.

Due to the M/G/1 models mentioned in Section 3.2, our formulation becomes a Mixed Integer Non-Linear Program. Moreover, it can be considered as an extension of [12] (because of, e.g., considering partitions for each component), which is a NP-hard problem, hence, we face a NP-hard MINLP problem. In the following, we describe the heuristic algorithm, based on a randomized greedy method, we developed to solve it (Algorithm 1).

Algorithm 1 Random greedy algorithm

1: **Input:** $\mathcal{I}$,$\mathcal{H}$, $\mathcal{J}$, DAG, $\mathbf{A}$, components demands, QoS constraints, system costs, MaxIter
2: **Initialization:** $BestSolution \leftarrow \emptyset$, $BestCost \leftarrow \infty$
3: **for** $m = 1, \ldots,$ MaxIter **do**
4: $\quad \mathbf{x} \leftarrow [0], \mathbf{y} \leftarrow [0], \widehat{\mathbf{y}} \leftarrow [0]$
5: $\quad$ Randomly pick a node j at each layer; set $x_j \leftarrow 1$
6: $\quad$ **for** $i \in \mathcal{I}$ **do**
7: $\quad\quad$ Randomly pick a deployment $c_s^i \in \mathcal{C}^i$ of component i
8: $\quad\quad$ **for** $h \in \mathcal{H}_s^i$ **do**
9: $\quad\quad\quad$ Randomly pick j s.t. $x_j = 1$ and $a_{hj}^i = 1$; set $y_{hj}^i \leftarrow 1$
10: $\quad\quad$ **end for**
11: $\quad$ **end for**
12: $\quad \widehat{y}_{hj}^i \leftarrow random[1, n_j] \cdot y_{hj}^i \ \forall i \in \mathcal{I}, \ \forall h \in \mathcal{H}_s^i, \ \forall$ VM j
13: $\quad$ **if** memory constraints are fulfilled **then**
14: $\quad\quad$ **if** local and global constraints are fulfilled **then**
15: $\quad\quad\quad$ ReduceVMClusterSize(j) for each VM j s.t. $x_j = 1$
16: $\quad\quad$ **end if**
17: $\quad$ **end if**
18: $\quad$ **if** $\langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle$ is feasible and $cost(\langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle) < BestCost$ **then**
19: $\quad\quad BestSolution \leftarrow \langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle$
20: $\quad\quad BestCost \leftarrow cost(\langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle)$
21: $\quad$ **end if**
22: **end for**
23: **if** $BestSolution \neq \emptyset$ **then**
24: $\quad$ **return** $BestSolution$
25: **else**
26: $\quad$ No feasible solution found
27: **end if**

The algorithm receives as input the compatibility matrix $\mathbf{A}$, the application DAG description with the performance demands, candidate device costs, local and global constraints, and the maximum number of iterations to be performed. First, we initialize the best solution and corresponding cost to infinity. At each iteration, we set matrices $\mathbf{x}$, $\mathbf{y}$ and $\widehat{\mathbf{y}}$ to zero (line 4), we randomly pick a device at each layer (line 5) and a deployment for each component, and we randomly assign each partition of the selected deployment to the selected devices according to the compatibility matrix $\mathbf{A}$ (lines 6-11). For each VM type j, we randomly chose the number of nodes between 1 and n_j, i.e., the maximum number of instances (line 12). This generates a solution $\langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle$ that satisfies the compatibility constraints. Then, we check the feasibility of memory constraints (line 13), and QoS constraints (line 14). If possible, we reduce the maximum number of selected VMs (line 15), preserving the feasibility of the current solution. At line 18, we check if the current solution improves the *BestSolution*, which is updated accordingly (lines 19-21). The best solution found, if any, is returned

at lines 23-27.

6 Experimental results

In this section, we present the numerical experiments that evaluate the performance of our Random Greedy algorithm. They were run on a Linux Ubuntu server with 2.2 GHz CPU Intel with 40 cores and 32 GB memory.

In a previous work [29] we compared the Random Greedy with an exhaustive search in simplified deployment scenarios allocating components only on edge and only on cloud demonstrating that our approach can identify the global optimum solution in small settings. Here, in Section 6.1, we compare the cost and performance of the Random Greedy algorithm with the solution obtained by *HyperOpt* [4], an open-source Python library. HyperOpt is based on Bayesian optimization algorithms over awkward search spaces, which may include real-valued, discrete, and conditional dimensions [4]. To make the Random Greedy performance analysis more robust, we consider also a *Hybrid Method* that exploits HyperOpt to improve an initial result provided by the Random Greedy algorithm (hence the Hybrid Method always obtains results at least as good as the Random Greedy). To quantitatively evaluate the different approaches, we define the percentage gain (denoted as *Cost ratio*) as follows:

$$\text{Cost ratio} = \frac{(OtherMethodCost - RandomGreedyCost)}{RandomGreedyCost} \times 100,$$

where $OtherMethodCost$ can denote the cost of HyperOpt or the Hybrid Method. We performed each experiment twice with $MaxIter = 1000$ and $MaxIter = 5000$. Note that, in each setting HyperOpt and the Hybrid Method always perform the same number of iterations of the Random Greedy.

Finally, Section 6.2 reports a scalability analysis aiming at assessing the effectiveness of our tool to tackle large-scale systems. This paper results dataset and the tool source code are available on Zenodo[1].

6.1 Use Case Comparison

In this section, we compare the cost and performance of the Random Greedy algorithm with the solution obtained by HyperOpt and the Hybrid Method on the use case described in Section 4. According to the use case, we consider five different components. The components can be placed across four computational layers, defined as follows:

- *Edge Resources* are included in two computational layers. The first hosts a drone with an entry-level compute board (cost: 4.55$/h), and one with a middle-level compute board (cost: 6.82 $/h). The second layer includes a PC and a GPU-based edge server (costs: 4.55$/h and 9.1$/h, respectively). Drones costs have been determined considering initial costs of

[1] `https://zenodo.org/record/5657843#.YYp8Ur3MLIE`

1000$ and 1500$, amortized over 2 years, and assuming that the application is executed 110 times per year. The initial costs of PC and edge server are of 1500$ and 3000$, respectively, amortized over 3 years (and the same number of executions on the field).

- *Cloud Resources* are all included in the third computational layer. We have considered G3 instances selected from the Amazon EC2 catalogue [2], powered by NVIDIA Tesla M60 GPUs equipped either with 4 vCPUs and 30.5GB of RAM (with a cost of 0.75$/h), or with 16 vCPUs and 122 GB of RAM (with a cost of 1.14$/h).

- *FaaS Resources* are selected from the AWS Lambda catalogue and are all included in the last computational layer. Their cost depends on the running component: the first configuration has a memory size of 4GB and a hourly cost of 0.06, 0.54, 0.16 and 0.96$/h when used to run components from C_2 to C_5, respectively. The second configuration has a memory size of 6GB; it is used only to run component C_5, with a cost of 0.83$/h. The expiration time is set to 10 minutes [3].

The detailed demands of the components and partitions is reported in Table 1. For what concerns the edge-to-cloud connectivity, we considered a 5G network with 1ms access delay [30] and 4Gb/s bandwidth.

[2] `https://aws.amazon.com/ec2/pricing/on-demand/`
[3] DOI: `https://doi.org/10.1109/TCC.2020.3033373`

Resource Type	**Demand [s]**																
	C_1	C_2			C_3						C_4			C_5			
	h_1	h_1	h_2	h_3	h_1	h_2	h_3	h_4	h_5	h_6	h_1	h_2	h_3	h_1	h_2	h_3	
Drone (Low-end Board)	1.26	1.20	0.60	0.60	-	-	-	-	-	-	-	-	-	-	-	-	
Drone (Mid-range Board)	1.00	0.79	0.40	0.40	-	-	-	-	-	-	-	-	-	-	-	-	
PC	-	1.00	0.50	0.50	5.00	2.50	2.50	2.00	2.00	2.00	-	-	-	-	-	-	
Edge Server	-	0.05	0.03	0.03	0.50	0.25	0.25	0.20	0.20	0.20	-	-	-	-	-	-	
G3 - 4vCPUs VM	-	0.02	0.01	0.01	0.11	0.05	0.05	0.04	0.04	0.04	0.03	0.02	0.02	-	-	-	
G3 - 16vCPUs VM	-	0.01	0.01	0.01	0.09	0.04	0.04	0.03	0.03	0.03	0.03	0.02	0.01	-	-	-	
FaaS - 4Gb (warm)	-	0.25	0.15	0.15	2.25	1.25	1.25	1.00	1.00	1.00	0.67	0.37	0.29	4.00	2.00	2.00	
FaaS - 4Gb (cold)	-	0.30	0.25	0.25	2.70	2.00	2.00	2.00	2.00	2.00	0.94	0.74	0.65	4.80	2.80	2.80	
FaaS - 6Gb (warm)	-	-	-	-	-	-	-	-	-	-	-	-	-	2.30	1.30	1.30	
FaaS - 6Gb (cold)	-	-	-	-	-	-	-	-	-	-	-	-	-	2.76	2.76	2.76	

Table 1: Demands of components and partitions on the compatible resources

We run the experiments for both light constraints (local and global constraints are equal to 20s) and strict constraints (the local constraint is equal to 2.5s and the global constraint is equal to 2s), with λ ranging in [0.1, 1] req/s with step 0.01 req/s. The methods cannot find any feasible solution for $\lambda > 0.47$ req/s in the strict constraints scenario and $\lambda > 0.94$ req/s in the light constraints scenario.

The cost comparison under light constraints is reported in Figure 4a and 4b, while Figure 4c shows the average running time of each method varying the number of iterations. The same is reported for the strict constraints setting in Figure 4d, 4e and 4f, respectively.

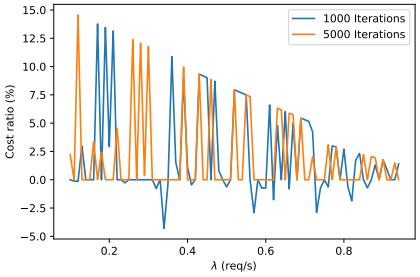
In order to fairly compare the performance, we run both Random Greedy and HyperOpt on one single core. However, note that our tool is based on a multi-threaded implementation that can leverage multi-cores. Unfortunately, HyperOpt is not able to exploit multiprocessing because it is a Bayesian sequential model-based optimizer. It can benefit from multi-cores only if run within Spark but in our single, 40-cores machine it was not possible to improve its performance.

As reported in Figure 4a, Random Greedy costs are up to 15% lower than those obtained by HyperOpt, while losses are infrequent and about up to 5% only for the 1000 iterations scenario. The Hybrid Method (Figure 4b) never improves Random Greedy for 5000 iterations, even if it obtains small gains (lower than 2%) at 1000 iterations. Under strict constraints, Random Greedy method always finds a feasible solution for $\lambda \leq 0.46$ req/s, while HyperOpt, even when running for 5000 iterations, cannot find any feasible solution for about 41% of scenarios (disconnected points in Figure 4d). However, it can be noticed that 1000 iterations are barely enough for Random Greedy, since in few cases, it loses up to 7% against the Hybrid Method (Figure 4e) and HyperOpt (Figure 4d), while it never loses when it runs for 5000 iterations.

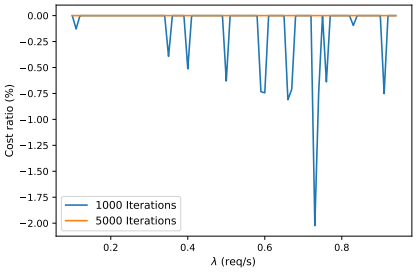
Note that, as reported in Figure 4c and Figure 4f, even if we run Random Greedy for 5000 iterations, the required time is at least one order of magnitude lower than the other methods running for 1000 iterations.

6.2 Scalability analysis

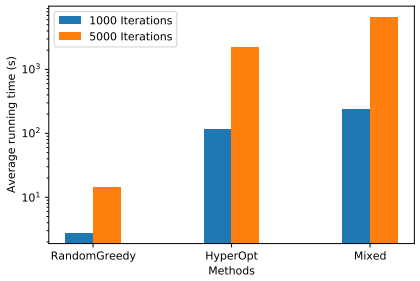
To evaluate the scalability of our approach, we considered four different scenarios at different scales as reported in Table 2. In the following, we report the average achieved by considering 10 random instances for each scenario. We randomly selected between 1 and 3 deployments for each component and between 1 and 4 partitions for each deployment. We also selected randomly, between 3 and 5, the maximum number of VMs of each type, while service demands were generated randomly in the range of $[1, 2]$s for drones, $[1, 5]$s for edge resources, $[0.5, 2]$s for VMs (as in [15]), and $[2, 5]$s for cold and warm FaaS requests (as in [31]). Note that, in the following scalability analysis, in order to consider the worst case in terms of average running time, we set local and global constraint thresholds to slightly larger numbers. Indeed, since each iteration terminates as soon as any constraint is violated, strict thresholds imply more frequent violations and also



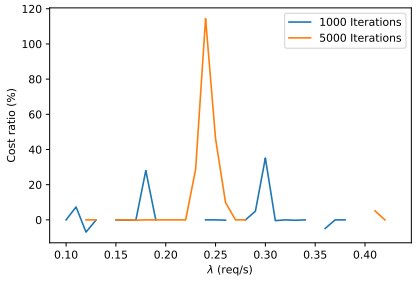
(a) Random Greedy vs. HyperOpt under light constraints



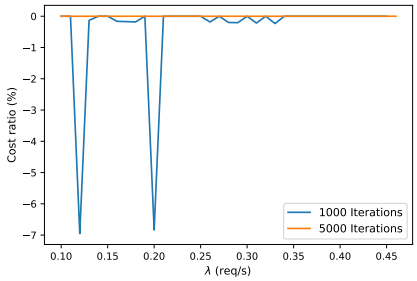
(b) Random Greedy vs. Hybrid Method under light constraints



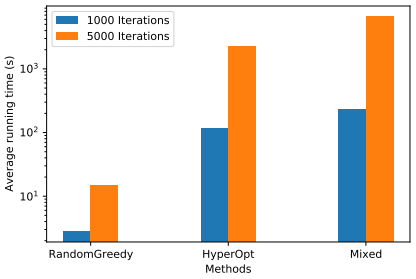
(c) Average running time of methods for light constraints.



(d) Random Greedy vs. HyperOpt under strict constraints



(e) Random Greedy vs. Hybrid Method under strict constraints



(f) Average running time of methods for strict constraints.

Figure 4: Comparison among the solutions obtained by Random Greedy, HyperOpt and the Hybrid method.

(a) Scalability parameters

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	
1	5	Drone: 2	Edge: 2	VM: 4	FaaS: 2	-	-	-	-	1, 1
2	7	Drone: 2	Edge: 4	VM: 4	VM: 4	FaaS: 2	-	-	-	2, 2
3	10	Drone: 2	Edge: 2	Edge: 2	VM: 4	VM: 4	VM: 4	FaaS: 2	-	3, 3
4	15	Drone: 2	Edge: 2	Edge: 2	VM: 4	VM: 4	VM: 4	VM: 4	FaaS: 2	4, 4

(b) Scalability performance evaluation

Scenario	**Avg. Exec. time for 1000 iterations (s), feasibility percentage**			**Avg. Exec. time for 5000 iterations (s), feasibility percentage**		
	RandomGreedy	*HyperOpt*	*HybridMethod*	*RandomGreedy*	*HyperOpt*	*HybridMethod*
1	3.93, 100	178.41, 80	333.61, 100	20.54, 100	3246.95, 90	9305.25, 100
2	9.37, 100	225.63, 100	407.66, 100	46.78, 100	4303.10, 100	11509.63, 100
3	15.02, 100	362.90, 100	494.15, 100	73.94, 100	7177.24, 100	14256.29, 100
4	14.80, 100	588.13, 60	611.71, 100	103.26, 100	12279.82, 40	16469.15, 100

Table 2: Scalability analysis

entail lower running times on average.

As it is shown in Table 2a, we considered problem instances including up to 15 components, 24 candidate nodes, 4 local and 4 global constraints. We set $\lambda = 0.11$ req/s and we replicated the experiment twice for $MaxIter = 1000$ and $MaxIter = 5000$. The average execution time and feasibility percentage across 10 instances is reported in Table 2b. Note that the maximum execution times for the Random Greedy are about 14s and 1.7 min with 1000 and 5000 iterations, while HyperOpt takes about 9.8 min and 3.5 hours, respectively. Moreover, HyperOpt cannot find a feasible solution for all instances (feasibility percentage about 83%).

Our approach can thus obtain significant improvements (with a factor of 162 and 120 for small and large scale, respectively) compared with HyperOpt in terms of average running time, which makes it suitable to tackle the component placement problem at design time.

As mentioned above, the iteration running time depends on the solution feasibility. For example in the largest case (15 components), the total time of an iteration that found a feasible solution is about 366 ms, where 2% of the total time is spent in creating the new solution and 98% in checking its feasibility, while the total time of an iteration that could not find a feasible solution is about 5 ms, where only 26% of the total time is spent in checking the feasibility of the solution (the iteration terminates as soon as any constraint is violated).

7 Conclusions

This paper proposes a randomized greedy approach to support application component placement and resource selection in computing continua at design time. Experimental results have shown that the approach can reach near-optimal solutions much faster than HyperOpt and as a result, it is useful to investigate multiple design choices spanning across heterogeneous system configurations including different edge devices and cloud based solutions. In our research agenda we plan to validate our solution on industry based case studies.

References

[1] M. Shirer. Worldwide "Whole Cloud" Spending Forecast, 2021–2025. `https://www.idc.com/getdoc.jsp?containerId=prUS48208321`, 2021.

[2] M. Peter and G. Timothy. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, USA, 2011.

[3] C. Hong and B. Varghese. Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms. *ACM Computing Surveys*, 52(5):1–37, 2019.

[4] Hyperopt: Distributed Hyperparameter Optimization. `https://github.com/hyperopt/hyperopt`.

[5] J. Bellendorf and Z. Ádám Mann. Classification of optimization problems in fog computing. *FGCS*, 107:158–176, 2020.

[6] M. Sriraghavendra, P. Chawla, H. Wuand S. S Gill, and R. Buyya. *DoSP: A Deadline-Aware Dynamic Service Placement Algorithm for Workflow-Oriented IoT Applications in Fog-Cloud Computing Environments*, pages 21–47. Springer Singapore, 2022.

[7] T. W. Pusztai, F. Rossi, and S. Dustdar. Pogonip: Scheduling asynchronous applications on the edge. In *IEEE CLOUD Proceedings*, 2021.

[8] C. Huang, X. Qiao, W. Lai, S. Dustdar, J. Zhang, and J. Li. Enabling dnn acceleration with data and model parallelization over ubiquitous end devices. *IEEE IoT-J*, pages 1–12, 2021.

[9] Z. Xu, W. Liang, W. Xu, M. Jia, and S. Guo. Efficient algorithms for capacitated cloudlet placements. *IEEE TPDS*, 27(10):2866–2880, 2016.

[10] A. Santoyo-González and C. Cervelló-Pastor. Latency-aware cost optimization of the service infrastructure placement in 5g networks. *JNCA*, 114:29–37, 2018.

[11] B. Tayebeh and Daniel G. Efficient placement of multi-component applications in edge computing systems. ACM SEC '17, 2017.

[12] W. Shiqiang, Z. Murtaza, and L. Kin. Online placement of multi-component applications in edge computing environments. *IEEE Access*, 5:2514–2533, 2017.

[13] A. Das, S. Imai, M. P. Wittie, and S. Patterson. Performance optimization for edge-cloud serverless platforms via dynamic task placement. *CoRR*, abs/2003.01310, 2020.

[14] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE TSE*, 33(6):369 – 384, 2007.

[15] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *IEEE/ACM SEC*, 2018.

[16] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE TPDS*, 30(9):2114 – 2129, 2019.

[17] Y. Yao, L. Rosasco, and A. Caponnetto. On early stopping in gradient descent learning. *Constr. Approx*, pages 289–315, 2007.

[18] H. Sivaraman, U. Kurkure, and L. Vu. Task Assignment in a Virtualized GPU Enabled Cloud. In *IEEE HPCS*, 2018.

[19] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[20] U. Tadakamalla and D. A. Menasce. Autonomic resource management for fog computing. *IEEE TCC*, pages 1–1, 2021.

[21] N. Mahmoudi and H. Khazaei. Performance modeling of serverless computing platforms. *IEEE TCC*, pages 1–1, 2020.

[22] Pricing calculator. `https://azure.microsoft.com/en-us/pricing/calculator/`.

[23] AWS . `https://aws.amazon.com/`.

[24] AWS Step Functions Pricing. `https://aws.amazon.com/step-functions/pricing/`.

[25] S. Risco, G. Moltó, D. M. Naranjo, and I. Blanquer. Serverless workflows for containerised applications in the cloud continuum. *Journal of Grid Computing*, 19(30):1–18, 2021.

[26] H. Sedghani, F. Filippini, and D. Ardagna. SPACE4-AI: A Design-time Tool for AI applications Resource Selection in Computing Continua (Technical Report). `https://www.dropbox.com/s/swce5b3iut4qhsz/Optimization_in_Fog_and_Cloud_Technical_Report.pdf?dl=0`, 2021.

[27] C. Lin and H. Khazaei. Modeling and optimization of performance and cost of serverless applications. *IEEE TPDS*, 32(3):615 – 632, 2021.

[28] Azure Logic Apps. `https://azure.microsoft.com/en-us/services/logic-apps/`.

[29] H. Sedghani, F. Filippini, and D. Ardagna. A randomized greedy method for ai applications component placement and resource selection in computing continua. In *IEEE JCC*, pages 1–6, 2021.

[30] 5G VS 4G: WHAT'S THE DIFFERENCE? `https://www.thalesgroup.com/en/worldwide-digital-identity-and-security/mobile/magazine/5g-vs-4g-whats-difference`.

[31] J. Manner, M. Endreß, T. Heckel, and G.Wirtz. Cold start influencing factors in function as a service. In *ACM/IEEE UCC Companion*, page 181–188. IEEE, 2018.