A Randomized Greedy Method for AI Applications Component Placement and Resource Selection in Computing Continua

Hamta Sedghani, Federica Filippini, Danilo Ardagna
Dipartimento di Elettronica Informazione e Bioingeneria,
Politecnico di Milano,
Milan, Italy
name.lastname@polimi.it

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, it 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.

1 Introduction

The importance and pervasiveness of Artificial Intelligence (AI) and Deep Learning (DL) are increasing dramatically in these years, with the AI software platform market expected to approach 11.8 billion in revenue by 2023, at a CAGR of 35.3% [1]. The Cloud Computing paradigm led this growth process, giving access to an ideally unlimited computational and storage power according to pay-to-go pricing models [2]. However, nowadays the accelerated migration towards mobile computing and Internet of Things (IoT) is determining an evolution of AI and big data applications. Indeed, data are generated by widespread end devices [3], which are characterised by growing computational capacity. Having

computing resources at the periphery of the network, novel applications can benefit from reduced latency, lower bandwidth requirements and increased energy efficiency and privacy protection. According to this trend, Edge intelligence, i.e., edge-based inferencing, is expected to become the foundation of many AI applications use cases, spanning from predictive maintenance to machine vision and healthcare.

Edge computing generates a fragmented scenario, where computing and storage power are distributed among devices with highly heterogeneous capacities. Therefore, component placement and resource allocation algorithms become 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.

These algorithms determine, at design time, the optimal deployment of all 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 underutilization.

This paper proposes a design-time tool to tackle the component placement problem and resource selection in the computing continuum, effectively addressing resource contention by adopting queueing theory to model application components response times. We developed an efficient randomized greedy algorithm, that identifies 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.

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 the use case we address in the paper, providing an example of the benefits our result would produce in real-life scenarios. Section 4 provides the problem statement and the algorithm we propose to tackle it. Experimental results are discussed in Section 5, while Section 6 describes the related works. Conclusions are finally drawn in Section 7.

2 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 requirements in Section 2.1, while we describe the computing continuum resources model, the network model and the system costs in Sections 2.2 and 2.3, respectively.

2.1 Application components model and QoS requirements

Applications are modeled as directed acyclic graphs (DAGs) (see Figure 1) whose nodes represent the different components. These are 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 [4, 5, 6], 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. The directed edge connecting components C_i and C_k 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 C_i to C_k. Furthermore, each C_i is characterized by a memory requirement (expressed in MB), and by a total load λ_i, which depends on λ and on the transition probabilities related to its predecessors (i.e., all components C_k such that an arc $\langle k, i \rangle$ exists in the application DAG, or, equivalently, such that p_{ki} is greater than zero).

For simplicity, we consider DAGs including only sequential execution and branches, since, as in [4], we assume that loops are unfolded (or peeled) while parallel execution is not supported for the time being. The set of all application components is denoted with $\mathcal{I}$.

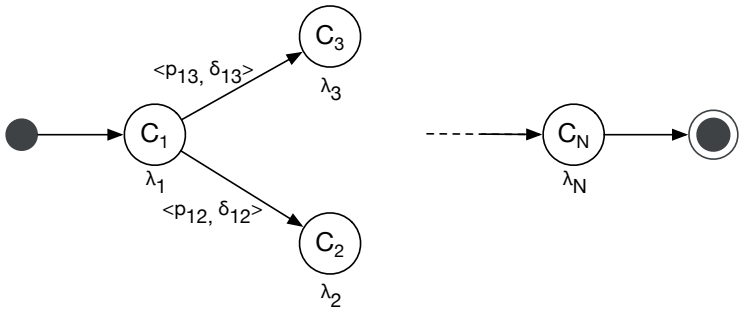


Figure 1: Directed acyclic graph for components.

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. Quality of Service 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*).

2.2 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.

We define different computational layers, including possibly heterogeneous resources. The first layer includes local devices generating data (such as drones, see Section 3). 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 ([7, 8, 9]), 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.

In order to compute the response time of all the executed components, we proceed 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 [10]).
- We model edge devices and VM instances as individual M/G/1 (single server multiple class) queues [11] 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 [12].
- 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., [11]) on the amount of data transferred, the network bandwidth and the access delay of the network domain. We neglect the network delay in cloud since all VMs and FaaS instances are executed in the same data center.

According to the results reported in [11] and [12], 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 [10] purposes.

2.3 System costs

Resources in the computing continuum are characterized by different costs:

- Costs of edge devices 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 [13, 14], while FaaS costs are expressed in GB-second [15, 16], 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., [17] and [18]) to account for the message passing and coordination. Other third party frameworks (e.g., [19] or [20]) 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 5.

3 A running example

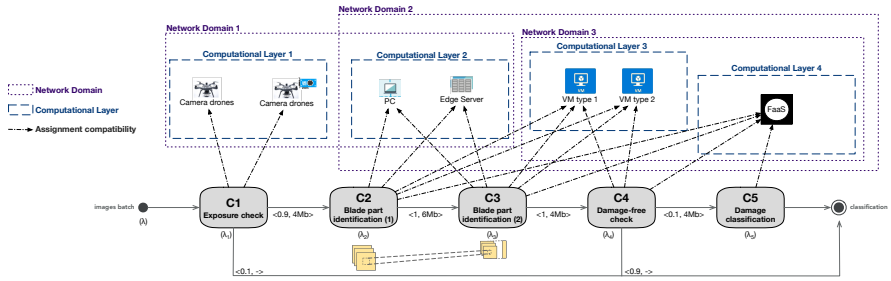


Figure 2: A Use case of identifying wind turbines blade damage.

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 Deep Neural Networks (DNNs) that can be deployed and 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 paradigm). The set of components is illustrated in Figure 2. They can be deployed overall on four layers, and the dotted arrows connecting each component to the different resources denote the corresponding compatibility.

As initial step, a drone with entry level or mid-range computation board, controlled remotely by a human operator, takes pictures of the wind turbine. These are composed by 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 of 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, the first layers of the DNN, responsible of the preliminary analysis of the images, are executed in C_2, while the mid and last layers, including the final classification, are executed in C_3. These components, especially C_3, require a significant amount of computing and storage power. However, executing them at the edge level may help in improving the process, if further data are needed to better identify the damages.

Images are then processed by two additional AI modules. C_4 is responsible of a *damage-free check*, i.e., of 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 of 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 aforementioned 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 information are transferred from edge to cloud resources through a, e.g., 5G network.

This scenario is characterized by both local and global QoS constraints, as described in Section 2.1. In particular, we prescribe that component C_5 must have a maximum response time of 2.5 seconds, while we enforce that the global response time of the first four components does not exceed 2 seconds.

4 Problem Statement and Solution

This section provides an overview on how we modeled the applications component placement and resource selection problem on heterogeneous edge and cloud resources. This is defined as a Mixed Integer Non-Linear Programming (MINLP) optimization problem, aiming at minimizing the deployment cost at

design time, while satisfying local and global Quality of Service (QoS) requirements (the formulation is omitted for space limit).

The main goal of our tool is to determine which kind of resource we should select at each computational layer, whether a component should be deployed on a given resource, and, in this case, if the assignment is compatible with memory constraints, used to determine the maximum number of components that can be co-located in each device, and QoS requirements.

Denoting with $\mathcal{J}$ the set of all resources in the computing continuum, in order to define the assignment decisions, namely to characterize which resources we are selecting at each computational layer and how components are assigned to the available devices, we introduce the following binary variables: y_{ij}, which, for all $C_i \in \mathcal{I}$ and $j \in \mathcal{J}$, is equal to 1 if and only if component C_i is deployed on node j, and x_j, which is 1 if and only if node $j \in \mathcal{J}$ is used in the final deployment. Furthermore, we introduce variables $\widehat{y}_{ij}$ to denote the number of VMs of type j assigned to any component C_i.

Due to M/G/1 models mentioned in Section 2.2, the problem becomes 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

Input: $\mathcal{I}$, $\mathcal{J}$, DAG, $\mathbf{A}$, components demands, QoS constraints, system costs, MaxIter
Initialization: $BestSolution \leftarrow \emptyset$, $BestCost \leftarrow \infty$
for $m = 1, \ldots,$ MaxIter **do**
    $\mathbf{x} \leftarrow [0], \mathbf{y} \leftarrow [0], \widehat{\mathbf{y}} \leftarrow [0]$
    Randomly pick a node j at each layer; set $x_j \leftarrow 1$
    for all $C_i \in \mathcal{I}$ **do**
        Randomly pick a node j s.t. $x_j = 1$ and $a_{ij} = 1$; set $y_{ij} \leftarrow 1$
    end for
    $\widehat{y}_{ij} \leftarrow random[1, n_j] \cdot y_{ij} \quad \forall\, C_i \in \mathcal{I},\ \forall$ Virtual Machine j
    if memory constraints are fulfilled **then**
        if local and global constraints are fulfilled **then**
            ReduceVMClusterSize(j) for each VM j s.t. $x_j = 1$
        end if
    end if
    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**
        $BestSolution \leftarrow \langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle$
        $BestCost \leftarrow cost(\langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle)$
    end if
end for
if $BestSolution \neq \emptyset$ **then**
    return $BestSolution$
else
    No feasible solution found
end if

The algorithm receives as input the compatibility matrix $\mathbf{A}$ (defined such that $a_{ij} = 1$ if and only if C_i can be executed on device j), the application DAG description with the performance demand, 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). Then, we randomly pick a device at each layer (line 5), and we randomly assign each component to the selected devices according to the compatibility matrix $\mathbf{A}$ (lines 6-8). For each VM, we randomly pick the number of nodes between 1 and n_j, i.e., the maximum number of available VMs of type j (line 9). 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 10), and QoS constraints (line 11). If possible, we tentatively try to reduce the maximum number of selected VMs (line 12), preserving the feasibility of the current solution. At line 15, we check if the current solution improves the *BestSolution*, which is updated accordingly (lines 16-18). The best solution found, if any, is returned at lines 20-24.

5 Experimental results

In this section, we present numerical experiments to evaluate the performance of our component placement approach. All experiments were run on a MacBook Pro with 2.4 GHz CPU Dual-Core Intel Core i7 and 16 GB memory. Specifically, the first set of experiments, detailed in Section 5.1, concerns the analysis of the use case presented in Section 3, while Section 5.2 reports a scalability analysis aiming at assessing the effectiveness of our tool to tackle large-scale systems[1].

5.1 Use case analysis

According to the use case described in Section 3, we consider five different components. The transition probabilities p_{ik} and the amount of data δ_{ik} transferred between components are reported in Figure 2. 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$/h, respectively). Drones costs have been determined considering initial costs of 1000$ 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 [13], 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).

[1]This paper results dataset and the tool source code are available at `https://zenodo.org/record/5091482`.

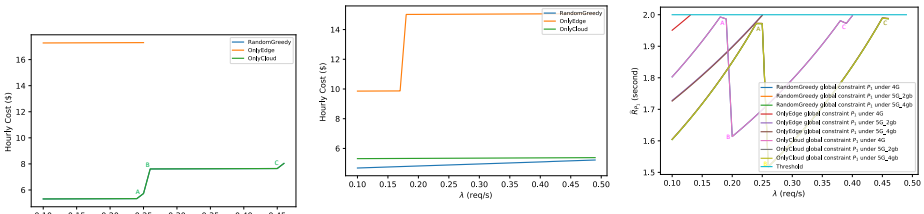
- *FaaS Resources* are selected from the AWS Lambda catalogue and are all included in the last computational layer. Their cost depends on the running component: the first configuration has a memory size of 4GB and a hourly cost of 0.06, 0.54, 0.16 and 0.96$/h when used to run components from C_2 to C_5, respectively. The second configuration has a memory size of 6GB; it is used only to run component C_5, with a cost of 0.83$/h. The expiration time is set to 10 minutes, as discussed in [12].

The demanding time of all components on the compatible resources are available in the Zenodo input files[1].

We have varied λ between 0.1 and 0.5 req/s, with step 0.01, to account for different workload scenarios. The first network domain is characterized by an access delay of 0.277 ms and a bandwidth of 150 Gb/s. For the second network domain, we have tested different settings: a 4G network, with a bandwidth of 20 Mb/s, and a slow or fast 5G network, with bandwidth equal to 2 or 4 Gb/s (the access delay is fixed to 0.277 ms). The bandwidth of the third network domain has been set to 100 Gb/s, while the access delay is 0, according to the results reported in [21]. We have imposed a local constraint on component C_5, that must have a response time below 2.5 s, and a global constraint on path $P_1 = \{C_1, C_2, C_3, C_4\}$, corresponding to a maximum response time of 2 s. Finally, Algorithm 1 performs 10000 iterations (MaxIter = 10000).

Our solution is compared with the results of an exhaustive search, where (to limit the execution time) some components are restricted to run on edge or on the remote cloud. In particular, in the *OnlyCloud* scenario, components C_2 and C_3 can run only on the cloud VMs, while they can run only on the edge in the *OnlyEdge* scenario.

The cost analysis, shown in Figure 3a and Figure 3b, is related to 5G@4Gb/s and it is slightly translated under other network settings. In all scenarios, when the incoming load is low, the best solutions use the entry-level compute board drone and the G3 4 vCPUs VM, which are slower and cheaper than the alternative configurations. Then, as λ increases, the response time along the path P_1 gets closer to the global constraint threshold incurring in a QoS violation (see Figure 3c). Therefore, the solution selects the G3 VM equipped with 16 vCPUs, which reduces the response time (point A in Figure 3c). If λ is further increased, the best solution steps back to G3 equipped with 4 vCPUs, but selects the drone with the mid-range compute board (point B in Figure 3c). Finally, the best solution adopts the faster and more expensive 16 vCPUs VM (point C in Figure 3c). When λ is about 0.41 req/s in the 4G scenario or 0.47 in the 5G scenarios, the response time of the path P_1 meets the threshold and no feasible solution can be found. As it can be noticed in Figure 3a and Figure 3c, our random greedy algorithm and *OnlyCloud* identify the same solutions. Indeed, according to components demands, resource costs and constraints, the *OnlyCloud* solution is the optimal solution because cloud VMs are both faster and cheaper than the available edge devices. This demonstrates that our algorithm converges to the global optimal solution. Vice versa, the *OnlyEdge* solution is expensive and slow and it is not even feasible for λ from 0.15 and 0.26 req/s in



(a) Cost variation of best solutions with increasing λ in 5G@4Gb/s scenario.

(b) Cost variation of best solutions with increasing λ for the constraints equal to 100s and 5G@4Gb/s scenario.

(c) Response time variation of path P_1 with increasing λ in three network scenarios. Performance overlap under the two 5G settings.

Figure 3: Experimental results

4G and 5G scenarios, respectively (see Figure 3a and Figure 3c).

Note that component C_5, which can run only on FaaS, is deployed on the high-end function configurations with 6GB of memory. This seems counter-intuitive because such configuration has a higher time-unit cost, but this deployment is proved to be more convenient, since the execution time of the component is decreased (and so the overall cost).

Costs increase linearly when λ ranges between 0.1 and 0.24 req/s, and between 0.26 and 0.45 req/s for the calls to AWS Lambda functions.

Finally, in Figure 3b, we increase the maximum response time of both local and global constraints to 100s in order to analyse how the cost changes under relaxed QoS requirements. In this scenario, the random greedy solution is cheaper than the *OnlyCloud* since it has more degrees of freedom and can execute components C_2 and C_3 on the FaaS configurations, which are slower but cheaper than VMs.

5.2 Scalability analysis

To evaluate the scalability of our approach, we considered five different scenarios at different scale reported in Table 1. We selected randomly, between 3 and 5, the maximum number of VMs of each type, while service demands were generated randomly in the range of $[1s, 2s]$ for drones, $[1s, 5s]$ for edge resources, $[0.5s, 2s]$ for VMs (as in [5]), and $[2s, 5s]$ for cold and warm FaaS requests (as in [22]). The local constraints threshold for C_i was set to $2 \cdot \max_j D_{ij}$, where D_{ij} is the demanding time of C_i on device j. Similarly, the global constraint threshold for path P was set to $2 \cdot \sum_{C_i \in P} \max_j D_{ij}$. We considered problem instances including up to 20 components, 20 candidate nodes, 8 local and 6 global constraints. We randomly generated 10 instances of each scenario at a given scale while we set $\lambda = 0.15$ req/s and $MaxIter = 10000$. The last column of Table 1 reports the average execution time across 10 instances. As it can be observed, the maximum execution time is of about 30 seconds, which makes our approach suitable to tackle the component placement problem at design time.

Table 1: Scalability analysis

Scenario	**#Components**	**#Nodes in Computational Layers (CL)**					**#Local constraints**	**#Global constraints**	**Avg. Exec. time (s)**
		CL_1	CL_2	CL_3	CL_4	CL_5			
1	5	Drone: 2	Edge: 2	VM: 2	FaaS: 2	-	1	1	10.39
2	10	Drone: 3	Edge: 4	VM: 3	VM: 3	FaaS: 2	4	2	15.27
3	15	Drone: 3	Edge: 4	VM: 4	VM: 4	FaaS: 2	6	4	29.76
4	20	Drone: 2	Edge: 3	VM: 4	VM: 2	FaaS: 2	8	6	23.86
5	20	Drone: 3	Edge: 5	VM: 5	VM: 5	FaaS: 2	8	6	30.52

6 Related Work

Components placement in computing continua has recently received a lot of attention from the research community. A recent survey is provided in [23] where authors propose a classification 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).

Among the proposals, [24], [25] and [26] are the closest to our approach. Authors in [24] 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 lower bound to estimate the performance of an online algorithm based on simple heuristic techniques such as iterative matching and local search. [25] 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, [26] 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.

7 Conclusions

This paper proposes a randomized greedy approach to support application component placement and resource selection in computing continua at design time. In our research agenda we plan to validate our solution on industry based case studies and to extend our approach to cope with multiple alternative configurations corresponding to different partitions of the same deep network.

Acknowledgements

The European Commission has partially funded this work under the H2020 grant n. 101016577 AI-SPRINT: AI in Secure Privacy pReserving computINg conTinuum.

References

[1] David Schubmehl. Worldwide Artificial Intelligence Software Platforms Forecast, 2019–2023. https://www.idc.com/getdoc.jsp?containerId=US44170119, 2019.

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

[3] Cisco Global Cloud Index: Forecast and Methodology, 2018–2023. https://www.cisco.com/c/en/us/solutions/collateral/serviceprovider/global-cloud-index-gci/white-paper-c11-738085.html.

[4] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Trans. on Software Engineering*, 33(6):369 – 384, 2007.

[5] Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *IEEE/ACM SEC*, 2018.

[6] Liang Bao, Chase Wu, Xiaoxuan Bu, Nana Ren, and Mengqing Shen. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE TPDS*, 30(9):2114 – 2129, 2019.

[7] Hari Sivaraman, Uday Kurkure, and Lan Vu. Task Assignment in a Virtualized GPU Enabled Cloud. In *IEEE HPCS*, 2018.

[8] Recommended GPU Instances. https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html.

[9] GPU optimized virtual machine sizes. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu.

[10] 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., Upper Saddle River, NJ, USA, 1984.

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

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

[13] Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

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

[15] AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[16] Azure Functions pricing. https://azure.microsoft.com/en-us/pricing/details/functions/.

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

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

[19] SCAR documentation. https://scar.readthedocs.io/en/latest/.

[20] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. Accelerated serverless computing based on GPU virtualization. *J. Parallel Distributed Comput.*, 139:32–42, 2020.

[21] Benchmarking Amazon VPC. https://aws.amazon.com/premiumsupport/knowledge-center/network-throughput-benchmark-linux-ec2.

[22] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. Cold start influencing factors in function as a service. In *ACM/IEEE UCC Companion*, page 181–188. IEEE, 2018.

[23] Julian Bellendorf and Zoltán Ádám Mann. Classification of optimization problems in fog computing. *Future Gener. Comput. Syst.*, 107:158–176, 2020.

[24] Tayebeh Bahreini and Daniel Grosu. Efficient placement of multi-component applications in edge computing systems. In *IEEE SEC*, 2017.

[25] Shiqiang Wang, Murtaza Zafer, and Kin K. Leung. Online placement of multi-component applications in edge computing environments. *IEEE Access*, 5:2514–2533, 2017.

[26] Anirban Das, Shigeru Imai, Mike P. Wittie, and Stacy Patterson. Performance optimization for edge-cloud serverless platforms via dynamic task placement, 2020.