SPACE4-AI: Design-time Management of AI applications Resource Selection in Computing Continua

APSC PROJECT

Author: **Randeep Singh, Giulia Mazzilli**

Advisor: Prof. Danilo Ardagna
Co-advisors: Federica Filippini, Hamta Sedghani
Academic Year: 2021-22

Abstract

We propose a `C++` implementation of SPACE4-AI, a tool meant to optimize the component placement and resource allocation for Artificial Intelligence (AI) applications at design-time. The need for this kind of tool starts from the rise of Edge computing, a model that deploys computing and storage power through several devices with different capacities.
This work will contribute to the AI-SPRINT project, a Research and Innovation Action co-funded by the European Commission, H2020 Programme [1].
SPACE4-AI was originally developed in `Python` [2], but the use of `C++` is crucial since it guarantees a much better execution time and it can serve as a starting point for the run-time version, where a good responsiveness is needed to cope with possible workload fluctuations. Thus, we designed a more efficient and scalable code that allows to quickly construct and check the feasibility of the solutions, and to parallelize the tasks on different threads.
Taking advantage of the new design, we tested our implementation against the original one, comparing not only the quality of the solutions to different system configurations, but also the performance in terms of execution time. We show that, while solutions' costs are comparable between the two versions, our implementation is able to compute them a hundred times faster, reducing execution times from four minutes to a couple of seconds for the largest problem instances.

Abstract in lingua italiana

Proponiamo un'implementazione in `C++` di SPACE4-AI, uno strumento volto all'ottimizzazione dello stanziamento di componenti e risorse per applicazioni AI nel design-time.
La necessità di questo tipo di strumento nasce dall'ascesa dell' "Edge computing", un modello che impegna numerosi dispositivi con diverse capacità per sfruttarne al meglio la potenza computazionale e la memoria.
Questo lavoro contribuirà al progetto AI-SPRINT, appartenente alle "Research and Innovation Actions" cofinanziate dalla Commissione Europea, Programma H2020 [1].
SPACE4-AI è originariamente scritta usando `Pyhton` [2] ma l'utilizzo di `C++` è cruciale perchè garantisce una maggiore reattività e può servire come punto di partenza per la "run-time version", in cui una buona ricettività è necessaria per stare al passo con le possibili fluttuazioni del carico di lavoro.
Dunque, abbiamo elaborato un'implementazione più efficiente e scalabile, che permette di costruire soluzioni e verificarne la praticabilità rapidamente, e di parallelizzare il carico di lavoro sfruttando diversi threads.
Ultimata la nuova versione del codice, abbiamo eseguito dei test per confrontarla con quella originale, concentrandoci non solo sulla qualità delle soluzioni trovate, ma anche sulle prestazioni in termini di tempo di esecuzione. Le varie analisi mostrano che, mentre i costi delle soluzioni ottenute dalle due versioni sono comparabili, la nostra implementazione è in grado di fornire risultati cento volte più velocemente, riducendo i tempi di esecuzione da quattro minuti ad un paio di secondi per le configurazioni più pesanti.

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 Application and Resource Models **3**
1.1 Application components model . . . 3
1.2 Local and global QoS constraints . . . 5
1.3 Resource general model . . . 5
1.4 Network model . . . 6
1.5 System costs . . . 6

2 A running example **9**
2.1 Maintenance and inspection of a wind farm . . . 9

3 Optimization problem formulation **13**
3.1 Selection variables and memory constraints . . . 13
3.2 Performance models . . . 14
3.2.1 Edge and Cloud . . . 15
3.2.2 Function as a Service . . . 16
3.3 Network transfer time . . . 17
3.4 Local and global QoS constraints . . . 19
3.5 System costs . . . 19
3.6 Optimization problem . . . 20

4 The solution algorithm **23**

5 The library **25**

5.1 Insights on the library structure . . . 25

5.1.1 Graph . . . 25

5.1.2 Resources . . . 27

5.1.3 System Data . . . 28

5.1.4 Solution Data . . . 30

5.1.5 Performance models . . . 32

5.1.6 System . . . 36

5.1.7 Solution . . . 36

5.1.8 Algorithm . . . 38

5.1.9 Other classes . . . 38

5.2 Parallelization . . . 40

5.3 Comments, issues and possible improvements . . . 40

5.3.1 Nested vectors . . . 40

5.3.2 Pseudo Random Number Generation . . . 41

5.3.3 Pybind11 and Parallelization . . . 41

6 Experimental results **43**

6.1 Costs comparison and scalability analysis . . . 43

6.2 Parallel algorithm performance . . . 46

6.3 Validation of the Random Greedy . . . 48

7 Conclusions and future developments **51**

Bibliography **53**

List of Figures **55**

List of Tables **57**

List of Symbols **59**

Introduction

The importance and pervasiveness of Artificial Intelligence (AI) are dramatically increasing in these years, together with an accelerated migration towards mobile computing and Internet of Things, where a huge amount of data is generated by widespread end devices. According to this trend, Edge intelligence is expected to become the foundation of many AI applications use cases, spanning from predictive maintenance to vision and healthcare.

Edge computing generates a fragmented scenario, where computing and storage power are distributed among multiple devices with highly heterogeneous capacities. In this framework, component placement and resource allocation become crucial to orchestrate in the most convenient way the physical resources of the computing continuum.

The component placement problem needs to be tackled in two different phases: design-time and run-time. Design-time choices aim at determining the optimal deployment of all application components on the candidate resources, satisfying different requirements and constraints. However, in practical applications, the workload (requests per second) expected at design-time is often subject to fluctuations due, for example, to the variations in the generated data volumes. For this reason, the initially designed component placement has to be continuously monitored and adapted online.

SPACE4-AI, initially developed in [2], exploits an efficient randomized greedy algorithm that identifies the placement of minimum cost 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. In this work we focus on the design-time version of the tool which, as a future development, will be extended to the run-time problem.
Our goal is to provide an efficient `C++` adaptation of the original implementation, written in `Python`, to reduce the computation times, especially when dealing with large scale systems. Experimental tests show that our library is able to compute solutions comparable to the ones of the previous implementation, but with execution times a hundred times smaller. Moreover, parallelization further yields a remarkable speed-up, which will be particularly useful for the run-time extension.

This report is organized as follows. Chapter 1 describes the application and computing continuum model. Chapter 2 provides an example of a real-life scenario that could exploit this tool. Chapter 3 reports the mathematical formulation of the problem. Chapter 4 introduces the algorithm to tackle it. Chapter 5 describes the main features of the `C++` implementation of SPACE4-AI. Finally, in Chapter 6 we discuss our results and perform a scalability analysis, while in Chapter 7 we draw our final considerations and comment on possible future developments.

1 | Application and Resource Models

In this chapter we introduce the general model developed for our application placement and resource selection tool.

1.1. Application components model

We model each application as a directed acyclic graph (DAG), where each node is a component of the application. We assume that a component is a `Python` function that can be run in a Docker container deployed either in an Edge device, in a Cloud Virtual Machine (VM) or using the Function as a Service (FaaS) paradigm. In our framework, components will be some Neural Networks used for image processing and classifications. The DAG includes a single entry point characterized by the input exogenous workload λ (requests/sec), and a single exit point. Moreover, we consider DAGs including only sequential execution and branches.

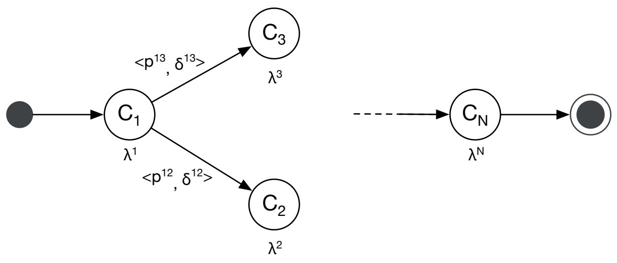


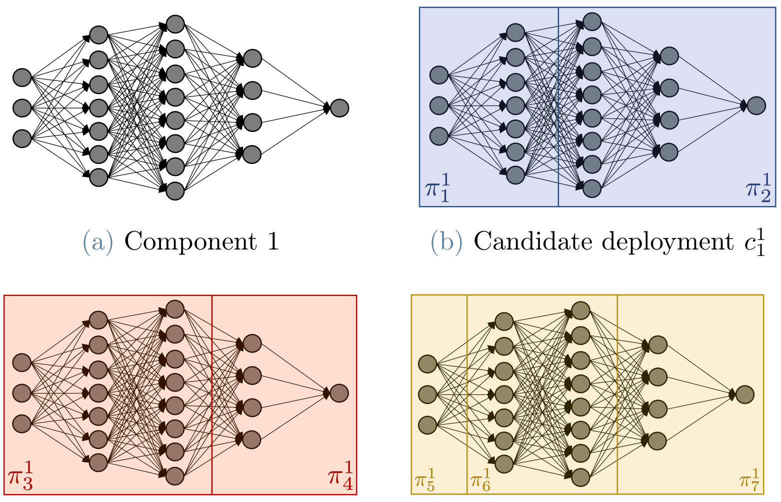
Figure 1.1: Directed acyclic graph for components

The directed edge from a node i to a node k is labelled with $< p^{ik}, \delta^{ik} >$, where p^{ik} is the transition probability between component i and component k, while δ^{ik} is the transferred data size.

We will denote with $\mathcal{I}$ the set of all application components. Each component $i \in \mathcal{I}$ is characterized by a load λ^i that depends on the exogenous workload λ and on the transition probabilities from the previous components (i.e., all components $k \in \mathcal{I}$ such that p^{ki} is greater than zero). If $Prec^i$ is the set of all components executed immediately before i we can define λ^i as:

$$\lambda^i = \sum_{k \in Prec^i \subseteq \mathcal{I}} p^{ki} \lambda^k. \tag{1.1}$$

Moreover components can be characterized by multiple candidate deployments. A deployment is made of different partitions of the Neural Network which identifies the component. We denote by $\mathcal{C}^i$ the set of all candidate deployments for component $i \in \mathcal{I}$. Each element $c^i_s \in \mathcal{C}^i$ is defined as $c^i_s = \{\pi^i_h\}_{h \in \mathcal{H}^i_s}$, where π^i_h denotes the Neural Network partition. The set $\mathcal{H}^i_s$ is defined as the set of indices h of all the partitions π^i_h in the candidate deployment c^i_s.



(a) Component 1 (b) Candidate deployment c^1_1

(c) Candidate deployment c^1_2 (d) Candidate deployment c^1_3

Figure 1.2: Example of AI application component with its candidate deployments

Fixed a component i, we also introduce an additional parameter $\tilde{p}^i_{h\xi}$, which defines the probability that partition π^i_ξ is executed right after partition π^i_h. This type of parameter is needed because we have to take into account mechanisms such as the early stopping of the Neural Networks. Indeed, in this framework, the early stopping mechanism entails that not all component partitions are necessarily executed, which dictates the necessity of defining the probability of actually moving from one to the other. To be more precise, early stopping can occur for both partitions and components: a component could quit its workflow before even starting and this would cause the termination of the whole AI application while, when a component's partition goes through early stopping, its workload is directly transferred to the next component. Similarly, we define $\tilde{\delta}^i_{h\xi}$ as the amount of data transferred from partition π_h^i to partition π_ξ^i. Consequently we can express the load $\tilde{\lambda}_h^i$ as:

$$\tilde{\lambda}_h^i = \begin{cases} \sum_{\pi_\xi^i \in Prec_h^i} \tilde{p}_{\xi h}^i \tilde{\lambda}_\xi^i & \text{if } Prec_h^i \neq \emptyset \\ \lambda^i & \text{otherwise} \end{cases} \tag{1.2}$$

where $Prec_h^i$ is the set of all partitions preceding π_h^i.
Finally, each partition π_h^i of component $i \in \mathcal{I}$ is characterized by the memory requirement $\tilde{m}_h^i$ (expressed in MB).

1.2. Local and global QoS constraints

We consider response time as the main performance indicator. R^i will be the response time of component $i \in \mathcal{I}$ and it will be greater or equal to the sum of the response times $\tilde{R}_h^i$ of all the partitions π_h^i in the deployment selected for component i.
We define execution paths as sequences of application components that start from the entry point and finish with the exit point of the DAG. $\mathcal{EP}$ is the set of all execution paths ep. A path P instead, is a set of consecutive components included in an execution path and we denote its response time with $\hat{R}_P$.
QoS requirements can be imposed on both the response time of a single component and of a path. Hence, we define as *local constraint* an upper bound threshold $\overline{LR}_i$ for the response time of a component $i \in \mathcal{I}$. The set of local constraints is characterized as follows:

$$\mathcal{LC} = \{\langle i, \overline{LR}_i \rangle : i \in \mathcal{I},\, R^i \leq \overline{LR}_i\}. \tag{1.3}$$

In the same way, a *global constraint* $\overline{GR}_P$ is a threshold for the response time of a path P:

$$\mathcal{GC} = \{\langle P, \overline{GR}_P \rangle : P \subseteq ep,\, ep \in \mathcal{EP},\, \hat{R}_P \leq \overline{GR}_P\}. \tag{1.4}$$

1.3. Resource general model

Computing continuum resources include devices located in the Edge, Cloud Virtual Machines (VMs) and Function as a Service (FaaS) configurations.
We define different layers that can include Edge or Cloud resources. The first layer includes local devices generating data (such as drones, as we will see in the next chapter) while the second layer is often located in the Edge and includes devices such as PCs or smartphones.

As for the VMs, we assume that they all come from a single provider catalogue and that the VMs selected at a certain layer are homogeneous and evenly share the workload due to the execution of one or more components. If VMs with GPUs are available, we limit our selection to the ones with just one GPU to optimize cost and performance.
Finally, we consider all the FaaS configurations in the same layer because each function will run on its own container independently. Note that the same component can be associated with multiple 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 Cloud devices by $\mathcal{J}_{\mathcal{C}} = \{E+1, ..., E+C\}$ and the set of all FaaS configurations by $\mathcal{J}_{\mathcal{F}} = \{E+C+1, ..., E+C+F\}$. As a consequence, $\mathcal{J} = \mathcal{J}_{\mathcal{E}} \cup \mathcal{J}_{\mathcal{C}} \cup \mathcal{J}_{\mathcal{F}}$ is the set of all candidate computing continuum resources.
For each partition π_h^i in all the candidate deployments c_s^i characterizing component $i \in \mathcal{I}$, we need to determine if a device $j \in \mathcal{J}$ can be used to execute π_h^i. To do this we introduce a compatibility matrix $\mathbf{A} = [a_{hj}^i]$ defined as:

$$a_{hj}^i = \begin{cases} 1 & \text{if partition } \pi_h^i \text{ can run on node } j \in \mathcal{J} \\ 0 & \text{otherwise.} \end{cases} \tag{1.5}$$

1.4. Network model

The communication between the different kinds of devices is enabled by several network domains. Being $\mathcal{D}$ the set of network domains, each network domain $d \in \mathcal{D}$ is characterized by the access time a^d and the bandwidth B^d. These are needed to compute the required time to support data transfers between two components that communicate with each other. For each network domain d, we define $\mathcal{ND}^d = \{l_1, ..., l_{n^d}\}$ as the set of layers included in the network domain.
Note that we can usually neglect the network delay in Cloud since all VMs and all functions instances are executed in the same data center.

1.5. System costs

The various resources involved in the computing continuum are characterized by different costs. Edge devices' costs can be estimated for the single run of the target application. For instance we can consider the sum of the investment costs amortized along the lifetime horizon of the device and the corresponding yearly management cost divided by the number of times the application is run over a year.
For what concerns the Cloud costs, the cost of VMs is considered to be a hourly cost, while

FaaS costs are expressed in GB-second and they depend on the memory size, the functions duration, and the total number of invocations. Moreover, when the FaaS paradigm is considered, an additional cost for each invocation might occur, depending on the provider.

2 | A running example

In this chapter we introduce a reference use case that will be quantitatively analysed in chapter 6.

2.1. Maintenance and inspection of a wind farm

To validate our work we investigate a use case related to the maintenance and inspection of a wind farm.
In this scenario, the identification of possible damages in the wind turbine blades is performed in a computing continuum, starting from images collected by drones.
The application software is characterized by multiple components, consisting of Deep Neural Networks that can be executed either locally (on the drone, on operators' PCs or on local Edge servers in the operators' van) or remotely (in a VM or through FaaS).
The set of components and resources is illustrated in the following figure.

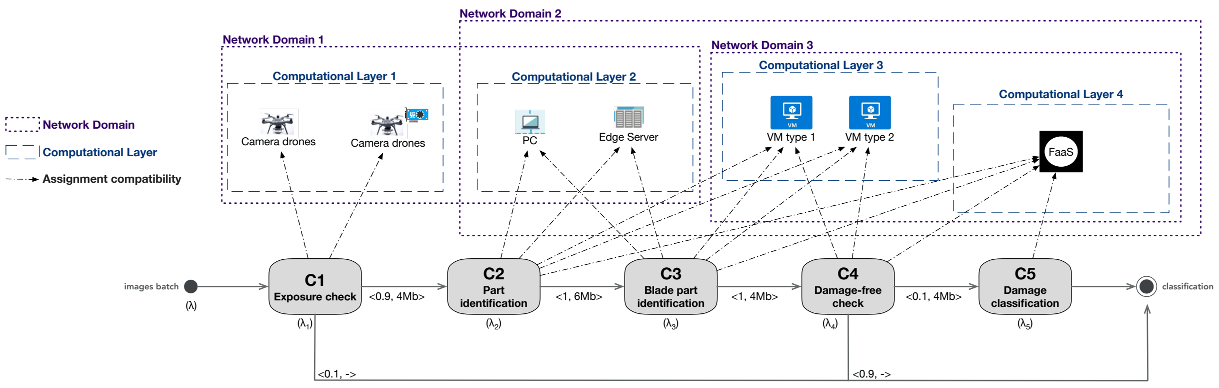


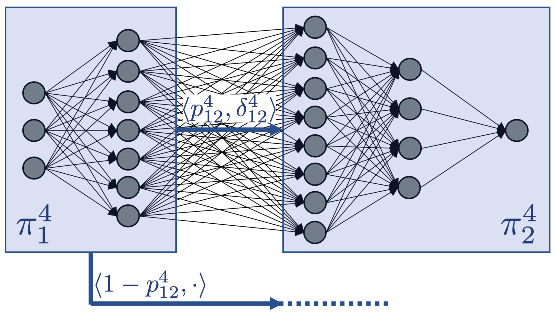
Figure 2.1: A use case of identifying wind turbines blade damage.

In the initial phase, a drone of the first layer, remotely controlled by a human operator, takes several thousands of pictures of the wind turbine. Each turbine is composed by three blades and pictures of each blade from four different angles must be collected in order to identify any kind of stress damage. At the first layer we can select a drone type with or without GPU on board.

Images are processed in batches and define the exogenous workload λ. Each batch is subject to a first *exposure check*, identified as component C_1, which determines if the image quality is sufficient for further processing. If not, the component, executed directly on the drone, triggers the acquisition of new images.
All pictures that pass the exposure test are then subject to a *damage free check*, implemented by C_2, which is run at the Edge level by a PC or a local server. The aim of this step is to provide an initial classification, identifying whether the inspected part is damaged or not.
After this second test, images are processed by two additional AI modules, C_3 and C_4, that are responsible of the blade part identification and of the object detection, that isolates the damaged portions by drawing boxes around them.
The execution of these components require a more significant amount of computational and storage power, hence they can be executed either at the Edge level, using a PC or the local server, or in the Cloud, if the Edge capacity has been saturated.
The last three steps are the following: first of all the images are post processed to better isolate the region of interest; then, damages are categorized according to their characteristics, with labels that identify the cause of the problem (such as cracks, lightning, etc.); finally, the severity of the damage is assessed according to five different levels, determining how urgent it is to intervene and fix the damage. These phases have heavy computational requirements, therefore they are always performed in the Cloud.
Until now we mentioned four computational layers:

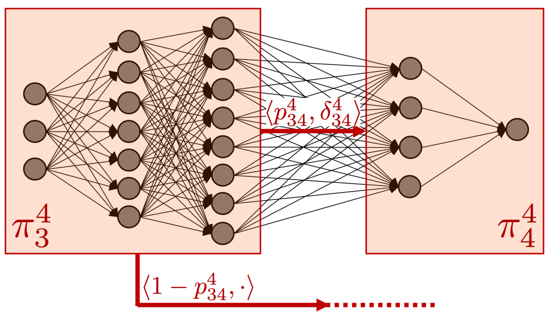
- the one involving camera drones;
- the one of Edge resources;
- the one including Virtual Machines;
- the one of FaaS.

These layers belong to three different network domains that characterize how data are transferred among different devices. Namely, drones and all Edge resources communicate through a Wi-Fi network; Cloud resources are connected via a fiber optic network, while information are transferred from Edge to Cloud resources through a 5G network.
All components can be characterized by multiple candidate deployments. In particular, we consider component C_4 to have two deployments, illustrated in the following image.



π_1^4
$\langle p_{12}^4, \delta_{12}^4 \rangle$
π_2^4
$\langle 1 - p_{12}^4, \cdot \rangle$

(a) Deployment c_1^4



π_3^4
$\langle p_{34}^4, \delta_{34}^4 \rangle$
π_4^4
$\langle 1 - p_{34}^4, \cdot \rangle$

(b) Deployment c_2^4.

Figure 2.2: Candidate deployments of component C_4

Both c_1^4 and c_2^4 are 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 will be higher in the second deployment, since most of the computation is executed in the first partition.

3 | Optimization problem formulation

This chapter provides the modeling of the applications component placement and resource selection problem on heterogeneous Edge and Cloud resources. This type of optimization problem is defined as Mixed Integer Non-Linear Programming (MINLP) and it aims at minimizing the deployment cost at design-time, while satisfying local and global constraints. In particular, this tool should be able to determine which resource is best at each computational layer, identify the appropriate deployment for each component, determine whether a partition included in that deployment should be executed on a given resource and check if the assignment is compatible with memory constraints and QoS requirements.

3.1. Selection variables and memory constraints

To define the assignment decisions we define the following binary variables:

$$x_j = \begin{cases} 1 & \text{if node } j \in \mathcal{J} \text{ is used} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

to characterize which resources we are selecting at each computational layer,

$$z_s^i = \begin{cases} 1 & \text{if deployment } c_s^i \text{ is selected for component } i \in \mathcal{I} \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

to determine which candidate deployment is selected for each component,

$$y_{hj}^i = \begin{cases} 1 & \text{if partition } \pi_h^i \text{ is deployed on node } j \in \mathcal{J} \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

to determine how the corresponding partitions are assigned to the available devices. Only one candidate deployment can be selected for each component, namely:

$$\sum_{s:c_s^i \in \mathcal{C}^i} z_s^i = 1 \qquad \forall i \in \mathcal{I}. \tag{3.4}$$

Once we have selected the deployment we have to assign its partitions to the most suitable resource. To ensure the compatibility of the selected device we must prescribe:

$$y_{hj}^i \leq a_{hj}^i \qquad \forall i \in \mathcal{I}, \forall h \in \mathcal{H}_s^i, \forall s : c_s^i \in \mathcal{C}^i \tag{3.5}$$

where a_{hj}^i is defined in equation (1.5).
Moreover a given partition π_h^i can be assigned to any device j only if the partition belongs to the selected deployment for component i, hence:

$$y_{hj}^i \leq z_s^i \qquad \forall i \in \mathcal{I}, \forall h \in \mathcal{H}_s^i, \forall s : c_s^i \in \mathcal{C}^i. \tag{3.6}$$

Finally, each partition can be assigned to a single device, which can be expressed as:

$$\sum_{j \in \mathcal{J}} y_{hj}^i = 1 \qquad \forall i \in \mathcal{I}, \forall h \in \mathcal{H}_s^i, \forall s : c_s^i \in \mathcal{C}^i. \tag{3.7}$$

Due to the memory limits of the resources, we need to bound the maximum number of partitions that can be co-located in each device depending on the memory. Let $\tilde{m}_h^i$ be the memory required by partition π_h^i. The total amount of memory required on device $j \in \mathcal{J}$ can be computed by summing $\tilde{m}_h^i$ over all partitions executed on j, hence we impose:

$$\sum_{i \in \mathcal{I}} \sum_{s:c_s^i \in \mathcal{C}^i} \sum_{h \in \mathcal{H}_s^i} \tilde{m}_h^i y_{hj}^i \leq M_j \qquad \forall j \in \mathcal{J} \tag{3.8}$$

where M_j is the memory limit of device j.

3.2. Performance models

For each computational layer l including Edge resources or Cloud VMs, we define the demanding time D_{hj}^{il} required to run the partition π_h^i on the 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. This definition, together with the queueing theory, is crucial to describe the performance models we adopt to estimate the response time of partitions executed on Edge resources or Cloud VMs. A different model will be adopted to compute the demanding time of partitions executed on FaaS configurations (see 3.2.2).

As a further note, observe that performance models can be divided in two categories:

- *static* performance models: the demanding time can be computed once and for all during system initialization, since it is independent from the the other partitions assignments and resource utilization;
- *dynamic* performance models: the demanding time depends also on other partitions and resources so it must be computed for each different solution configuration.

3.2.1. Edge and Cloud

We consider different types of Edge devices and VMs, and we denote by n_j the maximum number of devices of type $j \in \mathcal{L}^l \subseteq \mathcal{J}_{\mathcal{E}} \cup \mathcal{J}_{\mathcal{C}}$. We assume that several Edge nodes or VMs (of a single type j), evenly sharing the load, can be assigned to one partition. Hence, we introduce the variable $\hat{y}^i_{hj}$ to denote the number of devices of type j assigned to partition π^i_h.
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). Hence, the response time of a partition π^i_h deployed on a resource of any type j at any layer $l \in \mathcal{L}^l \subseteq \mathcal{J}_{\mathcal{E}} \cup \mathcal{J}_{\mathcal{C}}$ can be defined as:

$$\tilde{R}^i_h = \sum_{l:\mathcal{L}^l \subseteq \mathcal{J}_{\mathcal{E}} \cup \mathcal{J}_{\mathcal{C}}, j \in \mathcal{L}^l} \frac{D^{il}_{hj} \hat{y}^i_{hj}}{\hat{y}^i_{hj} - U_j} \tag{3.9}$$

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

$$U_j = \frac{\sum_{k \in \mathcal{I}} \sum_{\sigma: c^k_\sigma \in \mathcal{C}^k} \sum_{\xi \in \mathcal{H}^k_\sigma} \tilde{\lambda}^k_\xi D^{kl}_{\xi j} y^k_{\xi j}}{\hat{y}^i_{hj}}. \tag{3.10}$$

We further need to prescribe that the resources used to run the partition π^i_h are not saturated, i.e., that the equilibrium conditions for the M/G/1 queue hold:

$$\hat{y}^i_{hj} > 0 \Longrightarrow U_j < 1 \tag{3.11}$$

for all $i \in \mathcal{I}$, s such that $c^i_s \in \mathcal{C}^i$, $h \in \mathcal{H}^i_s$, $j \in \mathcal{L}^l \subseteq \mathcal{J}_{\mathcal{E}} \cup \mathcal{J}_{\mathcal{C}}$.
To conclude, since for all components $i \in \mathcal{I}$, for all s such that $c^i_s \in \mathcal{C}^i$, $h \in \mathcal{H}^i_s$, and for any $j \in \mathcal{L}^l \subseteq \mathcal{J}_{\mathcal{E}} \cup \mathcal{J}_{\mathcal{C}}$ the binary variable y^i_{hj} is equal to 1 when the partition π^i_h is executed on devices of type j, the equilibrium conditions expressed in (3.11) can be

expressed as:

$$\begin{cases} \hat{y}^i_{hj} + M(1 - y^i_{hj}) > 0 \\ \hat{y}^i_{hj} + M(1 - y^i_{hj}) > \sum_{k\in\mathcal{I}} \sum_{\sigma:c^k_\sigma\in\mathcal{C}^k} \sum_{\xi\in\mathcal{H}^k_\sigma} \tilde{\lambda}^k_\xi D^{kl}_{\xi j} y^k_{\xi j} \\ \hat{y}^i_{hj} \leq M y^i_{hj} \end{cases} \tag{3.12}$$

provided that M is a "large enough" constant.

From the definition 3.9, it is easy to see that this performance model is dynamic.

3.2.2. Function as a Service

Function as a Service (FaaS) is a type of Cloud computing service that provides a serverless paradigm to execute pieces of code, called functions, on a Cloud platform. The functions are executed in docker containers, so that users don't have to worry about managing the computing infrastructure. This tool has been proven to be cost effective, especially for dynamic workloads, because users only pay for the resources they actually use and not for the provisioned maximum capacity required by their system.
Accordingly with the docker container status, there are two types of requests: *hot requests* and *cold requests*. After the execution of a function, the FaaS paradigm keeps the Docker container running for a certain expiration time. If another function is invoked before the shutdown, the request is served immediately (hot request). If there are no hot containers available when the request arrives (cold request), the provider has to bring up a new container, causing a delay between the invocation and execution of the function. Users will not pay for the resources during this delay.
We denote by $d^{i,hot}_{hj}$ and by $d^{i,cold}_{hj}$ the execution time of hot and cold request of partition π^i_h on function configuration $j \in \mathcal{L}^l \subseteq \mathcal{J}_\mathcal{F}$. We also 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, on $d^{i,hot}_{hj}$, $d^{i,cold}_{hj}$, the expiration threshold and the arrival rate of function j, which is equal to the incoming workload $\tilde{\lambda}^i_h$ of the partition i running on the function. However, in general, the relation between the average execution time and the parameters is not trivial. Therefore, in this work we rely on an external library, namely `pacsltk` [3]. Eventually, we can define the response time of a partition π^i_h possibly running at the function $j \in \mathcal{L}^l \subseteq \mathcal{J}_\mathcal{F}$ as:

$$\tilde{R}^i_h = \sum_{l:\mathcal{L}^l\subseteq\mathcal{J}_\mathcal{F}, j\in\mathcal{L}^l} d^i_{hj} y^i_{hj}, \tag{3.13}$$

Note that this is a static performance model since the response time of each FaaS configuration is not directly influenced by the response times of the other FaaS deployed.

3.3. Network transfer time

To evaluate the global execution time of a component $i \in \mathcal{I}$ we also have to compute the network delay due to data transmissions between consecutive partitions. Thus, we define, for each pair of partitions (π^i_h, π^i_ξ) and 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 partitions executed on different devices in the same network domain d.
We define the network delay $\tilde{t}^{id}_{h\xi}$ as:

$$\tilde{t}^{id}_{h\xi} = \left(a^d + \frac{\tilde{\delta}^i_{h\xi}}{B^d} \right) \tilde{w}^{id}_{h\xi}. \tag{3.14}$$

By defining the set of resource indexes included in the network domain $d \in \mathcal{D}$ as:

$$\mathcal{UL}^d = \bigcup_{l \in \mathcal{ND}^d} \mathcal{L}^l, \tag{3.15}$$

we are able to characterize the variables $\tilde{w}^{id}_{h\xi}$, for all components $i \in \mathcal{I}$, for all s such that $c^i_s \in \mathcal{C}^i$, for all partitions π^i_h, π^i_ξ such that $h, \xi \in \mathcal{H}^i_s$, and for all network domains $d \in \mathcal{D}$, through the following conditions:

$$\begin{cases} \tilde{w}^{id}_{h\xi} \leq M^i_N \tilde{p}^i_{h\xi} \sum\limits_{r,v \in \mathcal{UL}^d} y^i_{hr} y^i_{\xi v} & \text{(3.16a)} \\ \tilde{w}^{id}_{h\xi} \leq 1 - \sum\limits_{j \in \mathcal{UL}^d} y^i_{hj} y^i_{\xi j} & \text{(3.16b)} \\ \tilde{w}^{id}_{h\xi} \geq \tilde{p}^i_{h\xi} \sum\limits_{r,v \in \mathcal{UL}^d} y^i_{hr} y^i_{\xi v} - \sum\limits_{j \in \mathcal{UL}^d} y^i_{hj} y^i_{\xi j}. & \text{(3.16c)} \end{cases}$$

In particular, inequality (3.16a) enforces that $\tilde{w}^{id}_{h\xi}$ must be equal to 0 if π^i_h and π^k_ξ are not consecutive partitions (expressed by the fact that $\tilde{p}^i_{h\xi}$ is 0) or if they are not executed on resources belonging to the same network domain, while it can be 1 if both conditions are satisfied (provided that M^i_N is a "large enough" constant). Inequality (3.16b) enforces that $\tilde{w}^{id}_{h\xi}$ is 0 if π^i_h and π^k_ξ are executed on the same device j. Inequality (3.16c) is complementary and it entails that $\tilde{w}^{id}_{h\xi}$ is equal to 1 when all the above conditions are satisfied, i.e. π^i_h and π^k_ξ are consecutive partitions executed on different devices belonging to the same network domain d.

In order 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:

$$t^{ikd} = \left(a^d + \frac{\delta^{ik}}{B^d}\right) w^{ikd}. \tag{3.17}$$

The data transmission between compontent i and k coincides with the one between the last partition of component i and the first partition of component k. Hence, we prescribe $w^{ikd} = 1$ if i and k are consecutive components whose last and first partitions are executed on different devices belonging to the same network domain d.
By introducing the additional boolean variables:

$$\alpha_h^i = \begin{cases} 1 & \text{if } \pi_h^i \text{ is the first partition of component } i \\ 0 & \text{otherwise,} \end{cases} \tag{3.18}$$

$$w_h^i = \begin{cases} 1 & \text{if } \pi_h^i \text{ is the last partition of component } i \\ 0 & \text{otherwise,} \end{cases} \tag{3.19}$$

we can characterize the variables w^{ikd} for all components $i, k \in \mathcal{I}$ and for all network domains $d \in \mathcal{D}$, by:

$$\begin{cases} w^{ikd} \leq M_N p^{ik} \sum\limits_{r,v \in \mathcal{UL}^d} y_{hr}^i w_h^i y_{\xi v}^k \alpha_\xi^k & \text{(3.20a)} \\ w^{ikd} \leq 1 - \sum\limits_{r,v \in \mathcal{UL}^d} y_{hj}^i w_h^i y_{\xi j}^k \alpha_\xi^k & \text{(3.20b)} \\ w^{ikd} \geq p^{ik} \sum\limits_{r,v \in \mathcal{UL}^d} y_{hr}^i w_h^i y_{\xi v}^k \alpha_\xi^k - \sum\limits_{r,v \in \mathcal{UL}^d} y_{hj}^i w_h^i y_{\xi j}^k \alpha_\xi^k & \text{(3.20c)} \\ \sum\limits_{s:c_s^i \in \mathcal{C}^i} \sum\limits_{h \in \mathcal{H}_s^i} w_h^i = 1 & \text{(3.20d)} \\ \sum\limits_{\sigma:c_\sigma^k \in \mathcal{C}^k} \sum\limits_{\xi \in \mathcal{H}_\sigma^k} \alpha_\xi^k = 1. & \text{(3.20e)} \end{cases}$$

Inequalities (3.20a), (3.20b) and (3.20c) follow the corresponding inequalities (3.16a), (3.16b) and (3.16c), since, for example, by equations (3.3), (3.18) and (3.19), $y_{hr}^i w_h^i = 1$ if π_h^i is the last partition of component i and it is executed on $r \in \mathcal{UL}^d$.
Inequalities (3.20d) and (3.20e) impose that the last partition of component i and the first partition of component k are unique.

3.4. Local and global QoS constraints

Given the definitions in equations (3.9), and (3.13), as well as the network transfer time defined in equation (3.14), we are able to compute the response time R^i of a component $i \in \mathcal{I}$ as:

$$R^i = \sum_{s:c_s^i \in \mathcal{C}^i} \sum_{h \in \mathcal{H}_s^i} \tilde{R}_h^i z_s^i + \sum_{d \in \mathcal{D}} \sum_{s:c_s^i \in \mathcal{C}^i} \sum_{h,\xi \in \mathcal{H}_s^i} \tilde{t}_{h\xi}^{id}. \tag{3.21}$$

The first term is the sum of the execution times of all the partitions in the selected deployment, while the second term represents the total transfer time among these partitions. The response time $\hat{R}_P$ of a path P, i.e. a sequence of components, can be defined as:

$$\hat{R}_P = \sum_{i \in P} R^i + \sum_{\substack{i,k \in P: i \neq k \\ d \in \mathcal{D}}} t^{ikd}, \tag{3.22}$$

where the first term computes the response time of all components belonging to the path while the last term is the network delay due to data transfer among different components. To make sure that QoS requirements of the applications are respected, we define global response time constraints for the different paths. Therefore, we set a threshold $\overline{GR}_P$ and prescribe:

$$\hat{R}_P \leq \overline{GR}_P \qquad \forall \langle P, \overline{GR}_P \rangle \in \mathcal{GC}. \tag{3.23}$$

We can define local constraints similarly by prescribing other thresholds for the response time of the single components:

$$R^i \leq \overline{LR}_i \qquad \forall \langle i, \overline{LR}_i \rangle \in \mathcal{LC}. \tag{3.24}$$

3.5. System costs

As mentioned in section 3.2, 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 configuration costs are estimated in GB-second. In order to compute these costs, we denote with T the overall time an application is active for a single run and we assume that T is less than one hour.

We compute the execution cost of all Edge devices and Cloud VMs as:

$$C = \sum_{j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}} c_j \overline{y}_j, \tag{3.25}$$

where c_j is either the amortized cost of the Edge device or the VM hourly cost, depending on the considered resource j, while $\overline{y}_j$ is the maximum number of running resources of type $j \in \mathcal{J}_{\mathcal{E}} \cup \mathcal{J}_{\mathcal{C}}$.

The GB-second unit cost for executing π_h^i on the function configuration $j \in \mathcal{J}_{\mathcal{F}}$ will be denoted with $c_{hj}^{F,i}$. Taking into account that FaaS total cost depends on the memory size, the functions duration, and the total number of invocations, we can define the execution cost of the function layer as follows:

$$C_F = \sum_{i \in \mathcal{I}} \sum_{s:c_s^i \in \mathcal{C}^i} \sum_{h \in \mathcal{H}_s^i} \sum_{j \in \mathcal{J}_{\mathcal{F}}} c_{hj}^{F,i} d_{hj}^{i,hot} y_{hj}^i \tilde{\lambda}_h^i T. \tag{3.26}$$

Note that $c_{hj}^{F,i}$ includes the cost of the used memory and that $d_{hj}^{i,hot}$ is inversely proportional to the memory $\tilde{m}_h^i$ allocated to the partition π_h^i.
According to some FaaS providers, we need to introduce a *state transition cost*, that we will denote with c^T, to model the additional charge for the message passing and coordination between two successive functions. However, if the management is supported by an architectural component, the state transition cost c^T is set to 0. Thus, without loss of generality, we formulate the total transition cost as:

$$C_T = \sum_{i \in \mathcal{I}} \sum_{s:c_s^i \in \mathcal{C}^i} \sum_{h \in \mathcal{H}_s^i} \sum_{j \in \mathcal{J}_{\mathcal{F}}} c^T y_{hj}^i \tilde{\lambda}_h^i T. \tag{3.27}$$

3.6. Optimization problem

The optimization model aims at finding the Edge/Cloud component placement that minimizes the total application execution costs. For convenience, all the parameters will be summarized in 7.
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:
(3.4), (3.5), (3.6), (3.7), (3.8), (3.12), (3.16a), (3.16b), (3.16c), (3.20a), (3.20b), (3.20c), (3.20d), (3.20e), (3.23), (3.24),
and

$$\sum_{j \in \mathcal{L}^l} x_j = 1 \qquad \forall\, l : \mathcal{J}_{\mathcal{E}} \cup \mathcal{J}_{\mathcal{C}} \subseteq \mathcal{L}^l \tag{P1b}$$

$$y^i_{hj} \leq x_j \qquad \forall\, i \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ h \in \mathcal{H}^i_s,\ j \in \mathcal{J} \tag{P1c}$$

$$\hat{y}^i_{hj} \leq \overline{y}_j \qquad \forall\, i \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ h \in \mathcal{H}^i_s,\ j \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{E} \tag{P1d}$$

$$y^i_{hj} \leq \hat{y}^i_{hj} \qquad \forall\, i \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ h \in \mathcal{H}^i_s,\ j \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{E} \tag{P1e}$$

$$\hat{y}^i_{hj} \leq n_j y^i_{hj} \qquad \forall\, i \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ h \in \mathcal{H}^i_s,\ j \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{E} \tag{P1f}$$

$$\begin{aligned} \tilde{p}^i_{h\xi} y^i_{hj} \leq \sum_{v \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{F}} y^k_{\xi v} \qquad & \forall\, i,k \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ \sigma : c^k_\sigma \in \mathcal{C}^k, \\ & h \in \mathcal{H}^i_s,\ \xi \in \mathcal{H}^k_\sigma,\ j \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{F} \end{aligned} \tag{P1g}$$

$$y^i_{hj} \in \{0,1\} \qquad \forall\, i \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ h \in \mathcal{H}^i_s,\ j \in \mathcal{J} \tag{P1h}$$

$$\hat{y}^i_{hj} \in \mathbb{N} \qquad \forall\, i \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ h \in \mathcal{H}^i_s,\ j \in \mathcal{J}_\mathcal{C} \tag{P1i}$$

$$\overline{y}_j \in \mathbb{N} \qquad \forall j \in \mathcal{J}_\mathcal{C} \tag{P1j}$$

$$\tilde{t}^{id}_{h\xi} \geq 0 \qquad \forall\, i \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ h \in \mathcal{H}^i_s,\ d \in \mathcal{D} \tag{P1k}$$

$$t^{ikd} \geq 0 \qquad \forall i,k \in \mathcal{I},\ d \in \mathcal{D} \tag{P1l}$$

$$\tilde{w}^{id}_{h\xi} \in \{0,1\} \qquad \forall\, i \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ h \in \mathcal{H}^i_s,\ d \in \mathcal{D} \tag{P1m}$$

$$w^{ikd} \in \{0,1\} \qquad \forall i,k \in \mathcal{I},\ d \in \mathcal{D} \tag{P1n}$$

$$\alpha^i_h \in \{0,1\} \qquad \forall\, i \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ h \in \mathcal{H}^i_s \tag{P1o}$$

$$w^i_h \in \{0,1\} \qquad \forall\, i \in \mathcal{I},\ s : c^i_s \in \mathcal{C}^i,\ h \in \mathcal{H}^i_s. \tag{P1p}$$

(P1b) guarantees that a unique device is selected at each Edge or Cloud layer. Constraints (P1c) bound the allocation of components to the devices. Constraints (P1d) bound the number of allocated VMs and Edge resources to the maximum number of running VMs or Edge resources of type j, while constraints (P1e) avoid running a component on Edge devices or VMs that are not assigned to it. Constraints (P1f) enforce that the number of Edge resources or VMs of type j assigned to a component is not greater than the number of available devices of that type. Constraints (P1g) guarantee that, if a partition π^i_h is executed on Cloud VMs or FaaS configurations, the consecutive partition π^k_ξ is not executed on Edge devices (assignments cannot move back from Cloud/FaaS to Edge). The term at the left-hand-side, indeed, is strictly positive when partition π^i_h is deployed on a device $j \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{F}$ and π_h^i and π_ξ^k are consecutive. In such cases, the sum at the right-hand-side has to be greater than a positive number smaller than 1, but since it is a sum of binary variables, of which at most one, by equation (3.7), can be 1. This entails $\sum_{v \in \mathcal{J}_\mathcal{C} \cup \mathcal{J}_\mathcal{F}} y_\xi^k = 1$, i.e., that there exist a Cloud VM or FaaS configuration such that π_ξ^k is executed on it. Finally, constraints (P1h)-(P1p) define the decision variables' domain.

4 | The solution algorithm

In this chapter we define a random greedy algorithm to find an approximate solution of the MINLP described in Section 3.6.

The input arguments of the algorithm are the compatibility matrix $\mathbf{A}$, the application DAG description with the performance demand, candidate deployments and partitions, candidate device costs, local and global constraints, the maximum number of iterations to be performed and the number of solutions to return (line 1).

The first step is to initialize solutions as an empty list of dimension k, namely the number of solutions to be returned (line 2). Then, at each iteration we start with a zero assignment for the matrices $\mathbf{x}$, $\mathbf{y}$, $\hat{\mathbf{y}}$ (line 4). Next, for each layer we randomly pick a device (lines 5-7). For each component we randomly pick a deployment of the component and assign each partition of the selected deployment to the selected devices according to the compatibility matrix $\mathbf{A}$ (lines 8-13).

For each Edge device and VM, we randomly pick the number of nodes between 1 and n_j (lines 14-16). In this way, we have generated a random initial solution ($\hat{\mathbf{y}}$).

The next step is to check the solution's feasibility by making sure that memory constraints, global and local constraints are satisfied (line 17).

If the solution is feasible, we tentatively try to reduce the maximum number of Edge devices and VMs currently in use (lines 18-20), preserving the feasibility of the solution. Then we compute the cost of the current solution, we add it to the *Solutions* list and we sort the solutions by cost, making sure that if k is exceeded the most expensive solution is discarded (lines 21-26).

Finally, we return the top k solutions with lower costs (lines 29-33).

Algorithm 4.1 Random greedy algorithm

```
1: **Input:** $\mathcal{J}_\mathcal{E}, \mathcal{J}_\mathcal{C}, \mathcal{J}_\mathcal{F}, \mathcal{L}, \mathcal{C}, \mathcal{H}, \mathcal{ND}, \mathbf{A}, \text{DAG}, \mathcal{LC}, \mathcal{GC}, \hat{c}, \overline{c}, \tilde{c}, MaxIter, k$
2: **Initialization:** $Solutions \leftarrow \emptyset$
3: **for** $m = 1, ..., MaxIter$ **do**
4:     $\mathbf{x} \leftarrow [0], \mathbf{y} \leftarrow [0], \hat{\mathbf{y}} \leftarrow [0]$
5:     **for** $l \in \mathcal{L}$ **do**
6:         Randomly pick a node $j \in \mathcal{L}^l$ and $x_j \leftarrow 1$
7:     **end for**
8:     **for** $i \in \mathcal{I}$ **do**
9:         Randomly pick a deployment $c^i_s \in \mathcal{C}$ of component $i$
10:        **for** $h \in \mathcal{H}^i_s$ **do**
11:            Randomly pick for partition $\pi^i_h$ a node $j$ such that $x_j = 1$ and $a^i_{hj} = 1$ and set
               $y^i_{hj} \leftarrow 1, \hat{y}^i_{hj} \leftarrow 1$
12:        **end for**
13:    **end for**
14:    **for** $\forall j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$ such that $y^i_{hj} = 1$ **do**
15:        $\hat{y}^i_{hj} \leftarrow random[1, n_j] \, \forall i \in \mathcal{I}, \forall h \in \mathcal{H}^i_s$
16:    **end for**
17:    **if** the solution is feasible **then**
18:        **for** $\forall j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$ such that $y^i_{hj} > 1$ **do**
19:            ReduceClusterSize($j$)
20:        **end for**
21:        Compute the solution cost
22:        $Solutions \leftarrow Solutions \cup \hat{\mathbf{y}}$
23:        Sort solutions by cost
24:        **if** length($Solutions$)>$k$ **then**
25:            discard the most expensive solution
26:        **end if**
27:    **end if**
28: **end for**
29: **if** $Solutions \neq \emptyset$ **then**
30:    Return the top $k$ solutions with lower costs
31: **else**
32:    No feasible solution found
33: **end if**
```

5 | The library

You can find the source code of the library we implemented as part of this project at the following link: `https://github.com/randosrandom/Space4AI`.

For the installation of the library, the management of the dependencies and the general usage, directly refer to the `README.md` file in the repository.

5.1. Insights on the library structure

In the following we provide a general description of the main data structures, design patterns and algorithms we decided to use in the library, focusing more on the motivations of the choices rather than implementation details. For the latter, please refer to the `Doxygen` documentation.

5.1.1. Graph

Component

Refer to `src/Graph/Component.hpp`.

For the class managing a single Component (as well as Partition and Deployment), directly consult the documentation on the repository. Indeed, there is nothing special to highlight, except that some constructors use universal references by H. Sutter for efficiency.

Directed Acyclic Graph

Refer to `src/Graph/Dag.hpp,.cpp`, `src/TypeTraits.hpp`.

To represent the Directed Acyclic Graph (DAG) modeling each AI-application we do not use any external library, but we implement the main structures ourselves. In particular, a DAG can be uniquely identified by a lower triangular *pseudo-transition matrix*

$$Q = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ q_{21} & 0 & 0 & \cdots & 0 \\ q_{31} & q_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ q_{N1} & q_{N2} & q_{N3} & \cdots & 0 \end{bmatrix}, \quad Q \in \mathbb{R}^{N \times N},\ N = |\mathcal{I}|$$

where,

$$q_{ij} = \mathbb{P}(\text{"transition from component j to component i"}) = p^{ji}, \quad i, j \in \{1, \ldots, N\}$$

and

$$\sum_{i=1}^{N} q_{ij} \leq 1,\ \forall i \in 1, \ldots, N$$

Note that, in contrast to canonical stochastic matrices, transition probabilities from each component are stored columnwise. Moreover, we called it a pseudo-transition matrix since it is not guaranteed that $\sum_{i=1}^{N} q_{ij} = 1$ due to the mechanism of early stopping (see 1.1).

In the code, we model such matrix as

```
using DagMatrixType = std::vector<std::vector<double>>;
```

Actually, this choice is not the most efficient one since we are storing a lot of useless zeros. However, for our objective, it is enough to use the chosen data structure since, for the cases we have analyzed, $N \leq 15$, and it may never exceed 100 also in the future. So, for the moment, we can just use simple nested `std::vector`.

Furthermore, there is also the need to associate each component to an index, and it is crucial to assign the index appropriately, following the order of the components in the DAG (if c_i may be executed before c_j, the index of c_i has to be less than c_j's one). To impose that, we can't rely on the order of "appearance" in the configuration file, since the JSON Data Interchange Standard define an object of such type as a unordered set of name/value pairs and, therefore, JSON parsers traverse the fields in alphabetical order. Thus, to find the right structure of the DAG from the configuration file independently from the components' names, we implement some specific methods, such as:

```
std::vector<size_t>
find_graph_order() const; // implemented in the class DAG
```

To store the relations from components' names to components' indexes we use ordered and unordered maps, namely:

```
/** Hash map from components' names to the assigned indexes */
std::unordered_map<std::string, size_t> comp_name_to_idx;

/** Ordered map from indexes to components' names.
*   It is useful when there is the need to
*   loop on components following the execution order.
*/
std::map<size_t, std::string> idx_to_comp_name;
```

5.1.2. Resources

Refer to `src/Resources.hpp`, `src/TypeTraits.hpp`.

As already mentioned, there are three types of Resources, namely Edge, Virtual Machines (VM) and Function as a Service (FaaS). In the library, we use an *enum* class to represent these different types and we create a class *template* which uses the enum class as the template parameter.

```
/** Enum class to identify different types of resources */
enum class ResourceType : size_t
{
 Edge = 0,
 VM = 1,
 Faas = 2,
 Count = 3, /**< total number of resource types */
};
```

```
/** Template class to manage different type of resources. */
template <ResourceType Type>
class Resource;

/** Specialization for FaaS */
template <>
class Resource<ResourceType::Faas>;
```

The template class `Resource` stores the main information about each device, such as name, memory, cost, number of cores and so on.

Then, we implement the class `AllResources` that stores as a single entity the set of all resources, namely what we have called $\mathcal{J}$ in the optimization problem statement. The resources are saved as vectors, one for each type.

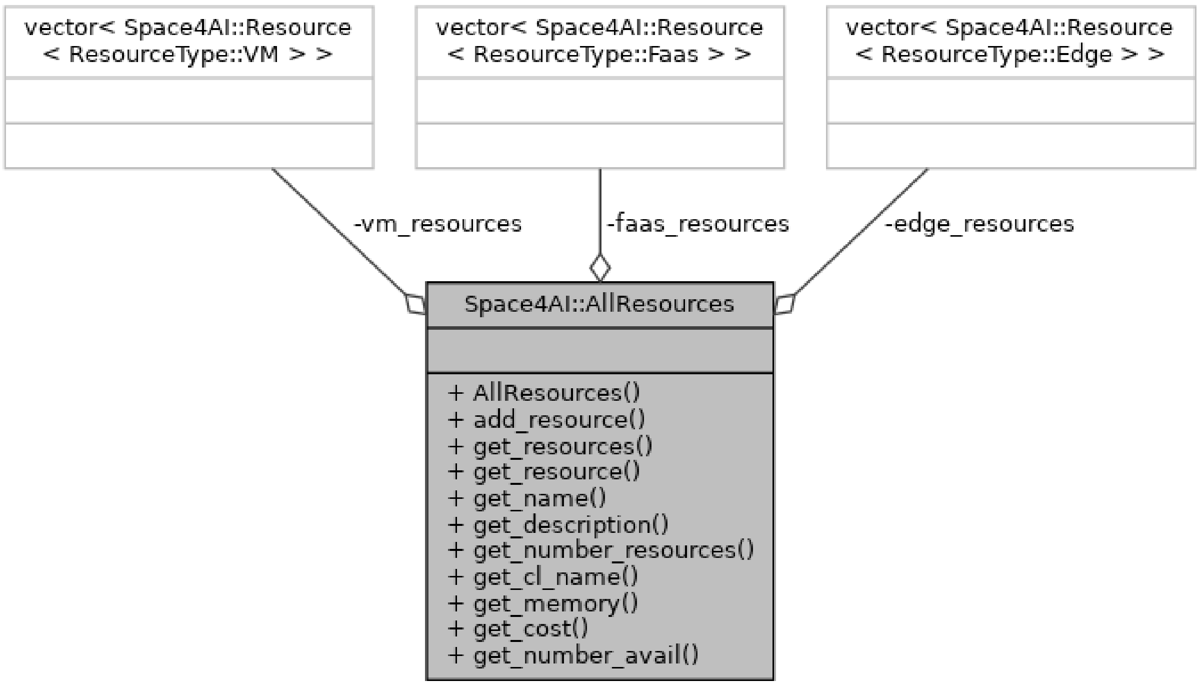


Figure 5.1: Collaboration diagram for *AllResources*

5.1.3. System Data

Refer to `src/System/SystemData.hpp,.cpp`.

`SystemData` is the class that stores everything about the configuration of the system but the performance models. So it contains the DAG and the resources, along with the constraints (local and global), the network technologies, the workload, the total time and so on. It has private constructors since the friend class `System` manages its lifetime. Mind that we can't simply incorporate `SystemData` to the general `System` class because the performance models objects, which will be part of the `System` class, require the data stored in `SystemData` to be fully defined beforehand.

Here, we describe a very important attribute of this class, namely the compatibility matrix. For the rest, resort to the documentation.

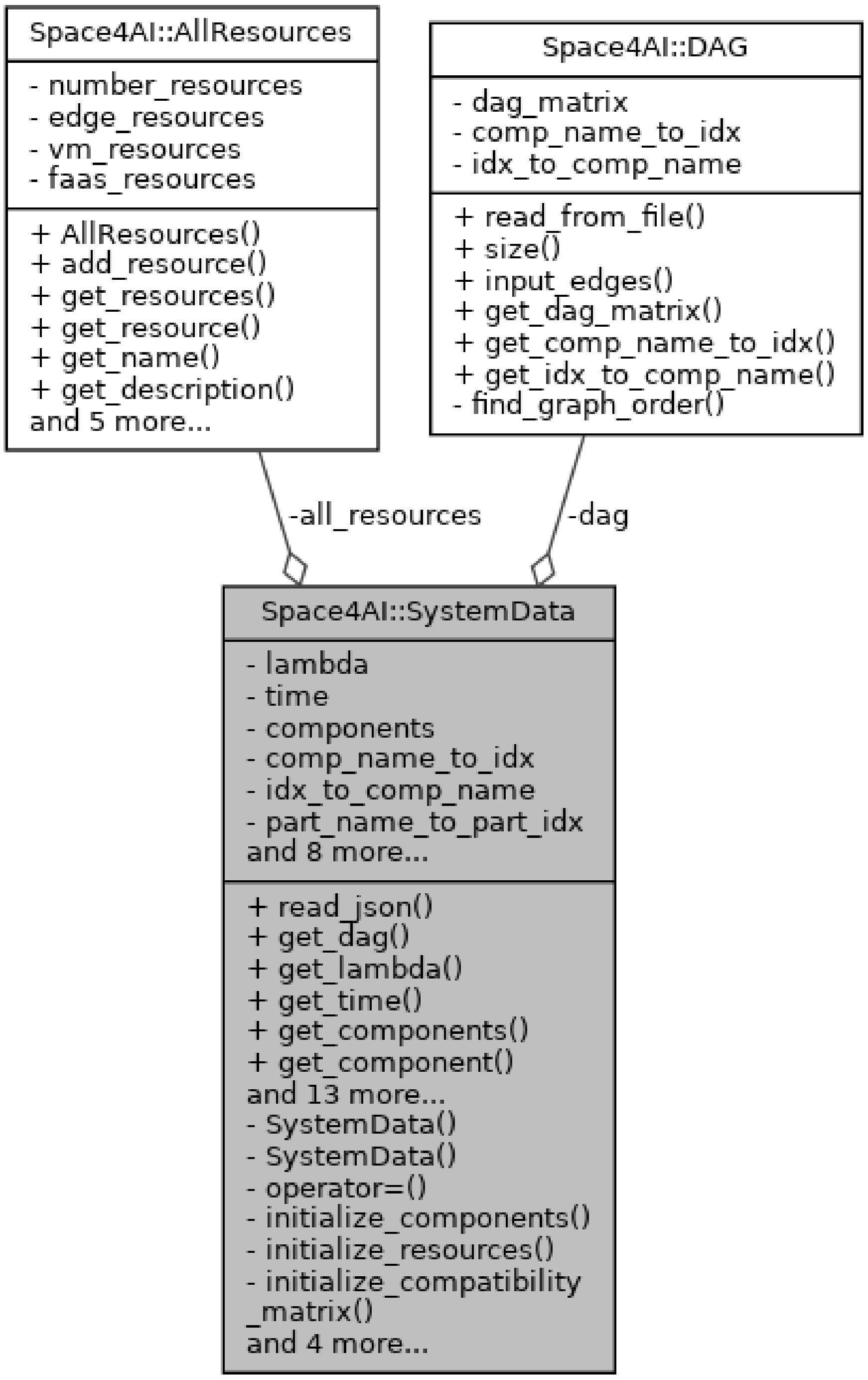


Figure 5.2: Collaboration graph for *SystemData* class

Compatibility Matrix

It is a four-dimensional structure coded as:

```
/** Compatibility Matrix of the system.
*
* Four-dimensional vector of booleans indexed by [i][j][k][l]
* storing true if partition k of component i is compatible with
* resource l of type j, false otherwise
*
*         i: index of the component
*         j: index of the resource type
*         k: index of partition
*         l: index of resource.
*/
using CompatibilityMatrixType =
    std::vector<std::vector<std::vector<std::vector<bool>>>>
```

We must admit that the chosen data structure, involving nested `std::vector` is not the ideal one. However, this was chosen according to the first optimization rule: *do not optimize anything*, at least prematurely. We started with the simplest solution that came to our mind but we are completely aware of the inefficiencies of such structure. However, since this "multi-vector" is not a "multi-array", in the sense that there is no guarantee that the entire multidimensional structure is rectangular (for each component there might be a different number of partitions, and for each type of resource we have a different number of resources available), no immediate alternative is available as flattening it is not so straightforward. So, a custom wrapper must be defined from scratch, but this will be a future development of the library. See also 5.3.1.

5.1.4. Solution Data

Refer to `src/TypeTraits.hpp`, `src/Solution/SolutionData.hpp`

`SolutionData` is a class that stores the main attributes identifying a solution. As for `SystemData`, the lifetime of this class is managed by the more general `Solution` class but, again, the former is necessary to implement the performance models.

In particular, the three key types used in the class are:

```
/** Type of the "y hat".
*   It stores the number of resources of a certain type,
*   running a certain component-partition couple.
*   Four-dimensional vector of size_t indexed by [i][j][k][l]
*        i: index of the component
*        j: index of the resource type
*        k: index of partition
*        l: index of resource.
*/
using YHatType =
    std::vector<std::vector<std::vector<std::vector<
    std::size_t
    >>>>;

/** Save the chosen partitions and utilized resources.
*   For each component (indexing the first vector) we save a vector
*   of tuples (partition index, ResourceType index, Resource index)
*   used on that component.
*/
using UsedResourcesOrderedType =
    std::vector<std::vector<
    std::tuple<size_t, size_t, size_t>
    >>;

/** For each [Resource Type, Resource idx] save the number of
*   instances of such resource utilized in the system.
*/
using UsedResourcesNumberType =
    std::vector<std::vector<size_t>>;
```

- `YHatType` defines the data structure to represent $\hat{\mathbf{y}}$; the very same observations done for the compatibility matrix 5.1.3 hold here as well. Again, refer also to 5.3.1.
- `UsedResourcesOrderedType` defines the data structure to save only the chosen partitions and utilized resources. It is used for efficiency, when there is the need to iterate only on the deployed resources and partitions, e.g., when calculating the total cost of the solution. Observe that the inner `std::vector` is sorted by parti-

tion index (actually the default `std::tuple<>::operator<()` is exploited), which is very useful for some operations like the computations of the network delays and the meeting of the constraint (P1g). Note that we don't use `std::set` instead of the inner `std::vector` since we do not have interleaved insertions/lookups, the number of insertions is way smaller than the number of lookups (indeed, once the random solution is built, no more insertions happen) and we don't need to insert in order but is sufficient to sort the `std::vector` once we have added all the elements. Therefore, as stated by Matt Auster [4], it is better not to use a `std::set` [4].

- `UsedResourcesNumberType` is used to store the number of activated instances of a specific resource of a specific type. It is not actually needed for FaaS because they are treated as single entities, so we can have at most a single instance for each configuration. However, for Edge and VM resources it is convenient to know the active instances without the necessity to loop on components.

5.1.5. Performance models

Refer to
`src/TypeTraits.hpp`,
`src/Performance/PerformanceFactory.hpp`,
`src/Performance/PerformanceModels.hpp,.cpp`,
`src/Performance/PerformancePredictors.hpp.in,.cpp`
`extras/Python/PACSLTK.py`

To implement the classes for the performance models, we use Polymorphism, and in figure 5.3 you can see the diagram. We used inheritance because it will be easier to add new performance models in the future although, at this point of the project, it is not strictly necessary to adopt this pattern.

For each triplet component-partition-resource we can have a different performance model (namely a different demand time to run the specific component-partition object on the specific resource). The associations are provided in the system configuration file, and are stored in the following data structure:

```
/** Predictors to compute the demand time.
*
*   For each partition and each component, running on
*   a specific resource of a specific type, we save an unique_ptr
*   to BasePerformanceModel.
*
```

```
*   Four-dimensional vector indexed by [i][j][k][l]
*        i: index of the component
*        j: index of the resource type
*        k: index of partition
*        l: index of resource.
*/
using PerformanceType =
    std::vector<std::vector<std::vector<std::vector<
    std::unique_ptr<BasePerformanceModel>
    >>>>;
```

`BasePerformanceModel` is an abstract class, and an *Object Factory* is deployed to create each performance model. Actually, the factory is just a plain function, with a bunch of if-else, since we necessitate a little flexibility to manage the different parameters and objects needed to create the various models.

```
/** Object factory for Performance Models.
*
*   \param model Model name of the model
*   \param perf_json  Piece of the system configuration json
*       file containing information about the specific
*       performance model.
*   \param system_data  As the name say
*   \param comp_idx   Component index
*   \param part_idx   Partition index
*   \param res_idx    Resource index
*
*   \return unique_ptr to the BasePerformanceModel abstract class
*/
inline
std::unique_ptr<BasePerformanceModel>
create_PE(
  const std::string& model, const nl::json& perf_json,
  const SystemData& system_data,
  size_t comp_idx, size_t part_idx, size_t res_idx)
```

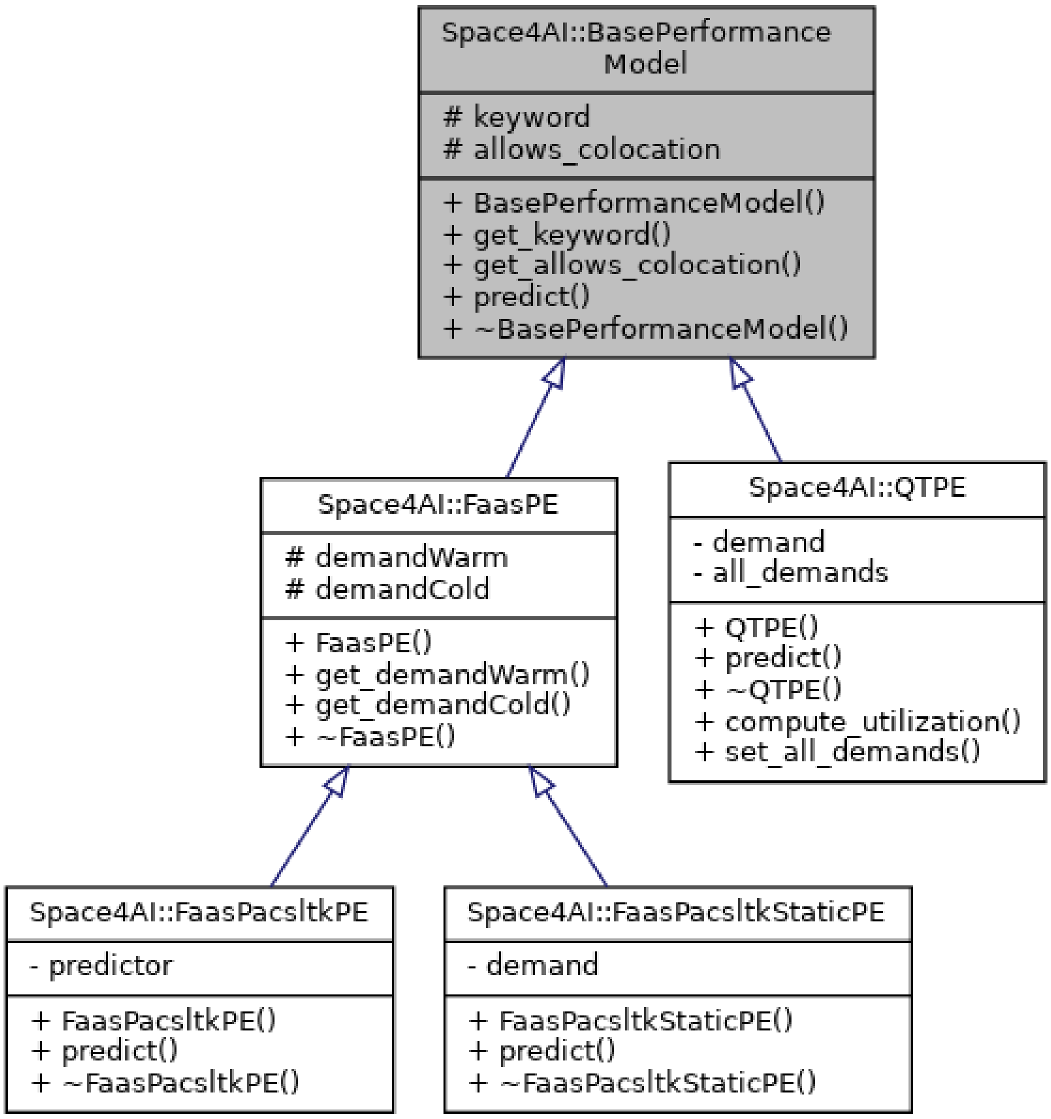


Figure 5.3: Inheritance diagram for *Space4AI::BasePerformanceModel*

Edge and Cloud

Edge and Cloud VM devices performance models are based on queuing theory, as already explained in 3.2, and they are implemented in the class `QTPE`. The special thing about this class is that it has a static member, namely `all_demands`:

```
/** Four-dim vector indexed by
*   [comp_idx][type_idx][part_idx][res_idx]) filled with Nans when
*   the specific resources running the specific component-partition
*   does not use QTPE.
*/
using DemandEdgeVMType =
  std::vector<std::vector<std::vector<std::vector<TimeType>>>>;

/* IN CLASS Space4AI::QTPE*/
/** Demand times of all Component-Partition objects that could be run
*   on the specific Resource of type Edge or Cloud VM.
*/
inline
static
DemandEdgeVMType all_demands = {};
```

It is used by the method `compute_utilization()`, since it needs to know not only the demand of the current partition, but also the demand times of all the other partitions running on the resource in analysis. Since, we are saving in a single structure all the possible demand times for all the possible partitions and resources, we can mark this attribute as static as it does not depend on the particular instance of the class. Note that, it is even optional to allow different instances of the class, but it is actually needed to preserve the interface, and to uniform the access to different types of models. Again, other choices are available, but for the moment the simplest structure possible is used.

Function as a Service

FaaS performance models are not straightforward to implement so we rely on an external library, namely `pacsltk` [3]. However, the package is implemented only for `Python`, so we have to embed the `Python` interpreter in our `C++` library. It is done using the `pybind11` library [5], which exposes `Python` types and functions using thin `C++` wrappers and makes it possible to conveniently call `Python` code from `C++`.

The script using the `pacsltk` library is written in a separate file, located in `extras /Python`. The `C++` wrapper is implemented in the class `Pacsltk`, implemented as Singleton. The location of the script is provided to the class using a `#cmake` MACRO in the file `src/Performance/PerformancePredictors.hpp.in`, properly configured in the `CMakeLists.txt`.

On top of this, we can define the interface for FaaS performance models, implemented in the abstract class `FaasPE`, publicly inherithed from `BasePerformanceModel`. Finally, we have the `FaasPacsltkPE` and `FaasPacsltkStaticPE`. The first is not strictly needed, it is just used for performance analysis. Indeed, since `pacsltk` defines a static performance model (as showed in 3.2.2), it is sufficient to compute the FaaS demands once and for all during the initialization of the system, and store it in the class, thus avoiding to call `Python` code during the construction of the solution. Mind that, the `pacsltk` package is very slow, and it would be the major library performance bottleneck if it was improperly used as a dynamic performance model.

5.1.6. System

Refer to `src/System.hpp,.cpp`

`System` incorporates all the objects needed to characterize an application defined in the system description *JSON* file, which is indeed parsed by this class through the help of the external library `JSON for Modern C++` [6].

It is the interface for the user to interact with the whole system, and it is the main class to deal with if there is the need to built a custom application using our library.

It stores a `SystemData` object along with the data structure for the performance classes.

System Performance Evaluator

Refer to `src/Performance/SystemPE.hpp,.cpp`

`SystemPE` is a class, of static methods only, that evaluates the performance of a system given a Solution (precisely a `SolutionData` object). It computes the local response times and the path response times, taking into account the network delays.

5.1.7. Solution

Refer to `src/Solution/Solution.hpp,.cpp`

`Solution` is the class that represents, as the name says, a solution to the system instance in

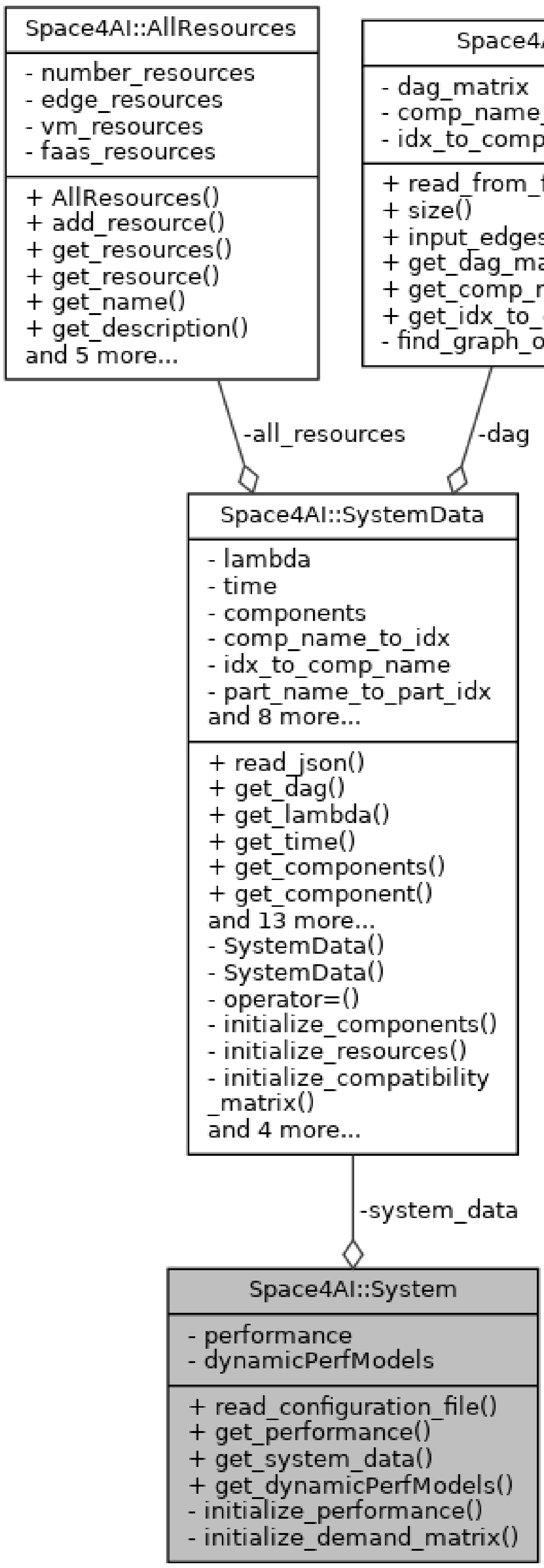


Figure 5.4: Collaboration diagram for *Space4AI::System* class

analysis. It saves a `SolutionData` object as well as the response times of the components and of the paths, and the total cost. Moreover, it has methods to check if all the various constraints are satisfied and to read/write solutions to file. Implementing such methods in an efficient way, especially those which check the fulfilment of all the constraints, is the key to obtain low execution times while running the algorithm.

We reiterate that we can not put together the two classes `SolutionData` and `Solution`, since to define `Solution` we first need a `System` object, which in turn has to have `BasePerformanceModel` already implemented. However, the latter must know how the basic data structure of the solution will be, which is why we first need to declare and define `SolutionData`, and a forward declaration is not enough.

Elite Result

Refer to `src/Solution/EliteResult.hpp,.cpp`

`EliteResult` is a class used to store a list of solutions sorted by cost. It is the object returned by the Random Greedy algorithm. The different solutions are saved in a `std::vector`, which is kept sorted and of the requested size.

5.1.8. Algorithm

Refer to `RandomGreedyDT.hpp,.cpp.in`

`RandomGreedyDT` is the class that implements the algorithm explained in 4. Along with `Solution` class methods, it is crucial to code the algorithm in the most efficient way possible. To achieve that, we used move semantic, avoided useless copies and properly resized data structures whenever possible. Loops are done in a cache-friendly way, but cache misses may happen due to nested `std::vector` (see 5.3).

5.1.9. Other classes

We list (non-exhaustively) the other important classes used in the library:

- `Logger, (src/Logger.hpp)`: used to print on terminal (or on file) useful logging messages, that can be turned off at any time, without the need of recompilation;
- `Chrono, (external/Chrono/)`: used to measure timings for the scalability analysis;
- `nlohmann::json, (external/nlohmann/)`: external class used to parse *JSON* files [6].

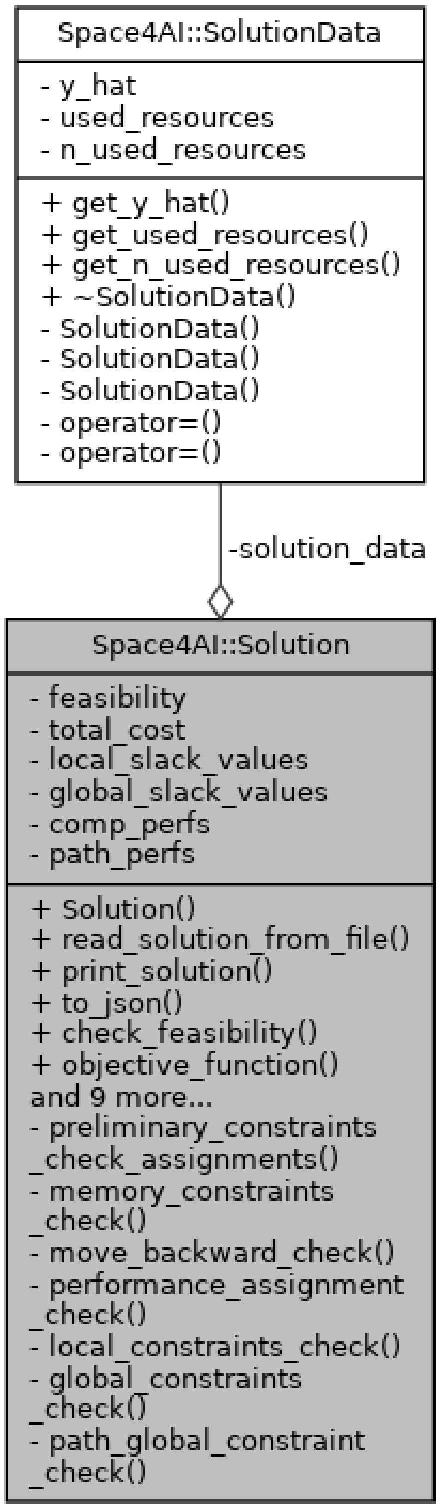


Figure 5.5: Collaboration graph of the class *Solution*

5.2. Parallelization

Our parallelization design is focused on the on the Random Greedy algorithm: it is sufficient to split the requested number of iterations (*MaxIter* in 4.1) among the different threads, since each iteration is independent from the others.

Given that the `System` object will be the same for each thread, we opted for `OpenMP` API to parallelize the algorithm, as it allows to easily share variables among threads. Moreover, in this way, it is also painless to compare solutions of the different processes and save them in a shared `EliteResult` object (of course using the `omp critical` directive), thus avoiding overheads when transferring solutions from one thread to another.

Global Interpreter Locker (GIL) issue

Since the library deals with `Python` code, we have to make an important note about the Global Intepreter Locker (GIL) [7]. In poor words, GIL is a mutex that allows only one thread to hold the control of the `Python` interpreter. This means that only one thread can execute `Python` code at any point in time. Of course, pure `Python` code can exploit the multi-processing, but this trick cannot be adopted when the actual parallelization is managed by someone else, like `C++` in our case. This is not really an issue for the performance models described in this report; nevertheless, if for any reason there is the need to compute demand times during the construction of the solution (e.g., due to the use of dynamic performance models implying `Python` calls), the parallelization of the algorithm can not be done for the part that deals with `Python` libraries, which however would most likely be the major performance bottleneck. Therefore, alternatives to embedding the `Python` interpreter in the library should be found, especially for heavy system instances and slow performance models.

5.3. Comments, issues and possible improvements

5.3.1. Nested vectors

As already mentioned, the most arguable choice we made in the library is the use of nested `std::vector`. Moreover, we also already said that no immediate replacement of such data structures can be found, but it is worth analyzing why nested `std::vector` are considered to be bad, at least when flattening is possible.

Besides the aesthetic aspect, technical criticalities are related to the facts that [8, 9]:

- they need to be allocated number-of-dimension times instead of one;
- random access can be affected by indirections, namely access like `v[i][j][k]` require three interdependent memory accesses, which is, in general, significantly slower than the indexing on 1D wrappers.
- processing data sequentially can be slow since the different vectors can be scattered around the memory, so there could be a lot of cache misses.

However, note that the overheads of using nested vectors are often negligible since there are many other things that dominate the total execution time of a program. Moreover, modern compilers can optimize things so no remarkable performance difference between the various data structure can be noted in particular cases [10].

Furthermore, in our specific case, another issue lies in the sparsity: despite no clear structure of the "multi-vectors" can be identified, since the system instances we may want to solve are endless, we could assume that many useless zeros will be saved (e.g., in the compatibility matrix). Nevertheless, in the cases we have analyzed the total number elements is in the order of 10^7, so it is not a big deal.

5.3.2. Pseudo Random Number Generation

In our algorithm to generate random numbers we employ the `std::mt19937` engine, and we use independent `std::uniform_int_distribution` to sample the integers in the wanted intervals. Nonetheless, to avoid sampling so many times, a sort of Quasi Monte Carlo method could be considered, through the use of the Sobol sequences. However, a straight application of the latter would be impossible due to the prohibitive dimensions of the overall sampling space. Therefore, an idea could be to exploit the Sobol sequences (in a suitable dimension to be chosen) as if they were random numbers generated by an engine. Of course, in this way we will loose independence between the samples, but for an adequate number of iterations we could expect to explore the sample space quite exhaustively, saving time drawing new samples each time.

5.3.3. Pybind11 and Parallelization

As alredy said, the part of `Python` calls in the algorithm cannot be parallelized, which is pretty limiting. Therefore, alternatives to embedding the `Python` interpreter in the library must be found. One idea could be to implement the performance models implying `Python` calls as Web Service [11], and integrate them in the library. Of course, Web Service would be less efficient than `pybind11`, but the first would naturally allow the parallelization since each request would be managed independently by the Web server. So the trade-off could be to use `pybind11` for the serial runs and Web Service for the parallel ones.

6 | Experimental results

In this chapter we present the numerical results, in terms of cost and execution time, we obtained with our Random Greedy algorithm and we compare it with the original `Python` implementation. The experiments were run in a Docker container, on a Linux Ubuntu server with Intel Xeon Silver 4114 2.20GHz CPU, 20 cores and 64 GB RAM.

Section 6.1 reports a scalability analysis aiming at assessing the effectiveness of this tool to tackle large-scale systems, and the improvement of the execution times with respect to the `Python` version.
In section 6.2 we test the parallelized algorithm and evaluate the speedup that can be reached.
Lastly, in section 6.3, we validate the solutions provided by our he algorithm with a state-of-the-art solver based on Bayesian Optimization, namely *HyperOpt* [12].

6.1. Costs comparison and scalability analysis

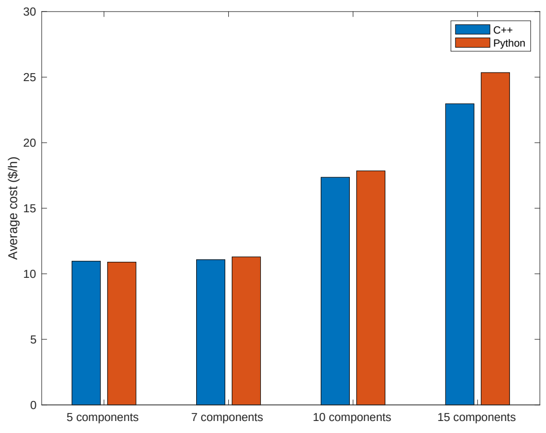
To compare the costs of the solutions obtained by our `C++` Random Greedy with the ones of the `Python` version, and to perform a scalability analysis, we considered four different scenarios at different scales, as reported in Table 6.1.
We randomly selected between 1 and 3 deployments for each component 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 and $[2, 5]$ s for cold and warm FaaS requests, as in other literature proposals and as reported in [2].
We considered problem instances including up to 15 components, 24 candidate nodes, 4 local and 4 global constraints. We set $\lambda = 0.10$ req/s and we replicated the experiment twice for $MaxIter = 500$ and $MaxIter = 5000$.
Figure 6.1 shows the comparison between the costs of the solutions generated by the the two versions. The reported costs have been achieved by considering the average across 10 random instances for each scenario. At a first glance the results seem to be comparable, with minimal differences, dictated by the randomness of the algorithm. This observation

is confirmed by Figure 6.2, where we can see the `C++` percentage cost increment for each scenario, computed as follows:

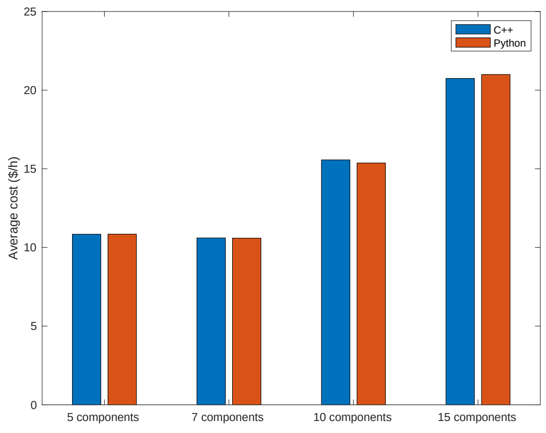
$$\texttt{C++}\text{ cost increment} = \frac{(\texttt{C++}AverageCost - \texttt{Python}AverageCost)}{\texttt{Python}AverageCost} \times 100.$$

Indeed we notice that, for 5000 iterations (Figure 6.2b), the percentage cost increment, both positive and negative, never exceeds 1.5%. For 500 iterations (Figure 6.2a) the cost increment (and decrement) does not go above 3%, except for the 15 components scenario where we see a more noticeable reduction of the costs (-9.38%). Keeping in mind that these results have a strong random component, we repeated the experiments a reasonable amount of times and we noticed that the 500 iterations - 15 components case always has an exceptional decrease of the costs.

For what concerns the execution time of the algorithm, in Table 6.2 we summarized the results of `Python`, `C++` serial and `C++` parallel (using 8 threads), considering the average across the same random instances mentioned above. We repeated the experiments for 500 and 5000 iterations. Note that, in the following analysis, to consider the worst case in terms of average running time, we set local and global constraints thresholds to large numbers. Indeed, since each iteration terminates as soon as any constraint is violated, strict thresholds imply more frequent violations and also entail lower running times on average.
The maximum execution times for the serial `C++` Random Greedy are about 0.124 s and 1.24 s with 500 and 5000 iterations, while the `Python` version takes about 19.35 s and 4 min, respectively. Moreover, we report the running time of the parallel version (using 8 threads) which takes at most about 27 ms and 190 ms for 500 and 5000 iterations respectively.
Thus, our `C++` implementation can obtain huge improvements compared to `Python` in terms of average running times, with a factor of 189 and 195 for the serial version at small and large scale respectively (5000 iterations case).

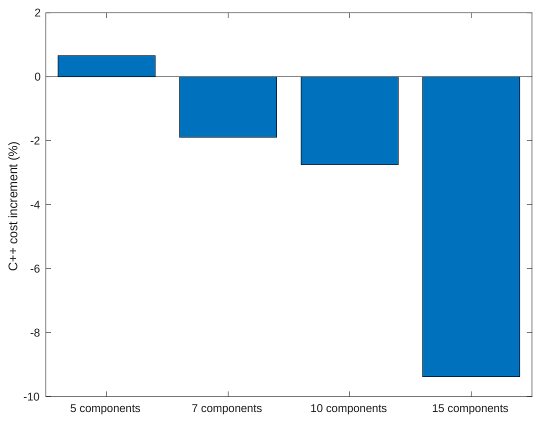


(a) 500 RandomGreedy iterations.

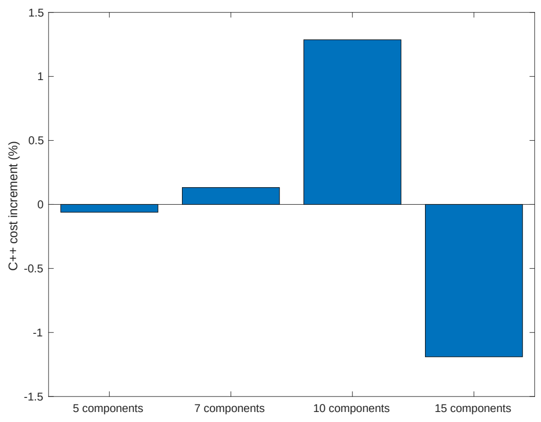


(b) 5000 RandomGreedy iterations.

Figure 6.1: Average costs comparison for each scenario.



(a) 500 RandomGreedy iterations.



(b) 5000 RandomGreedy iterations.

Figure 6.2: C++ relative cost increment for each 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	
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

Table 6.1: Scalability parameters

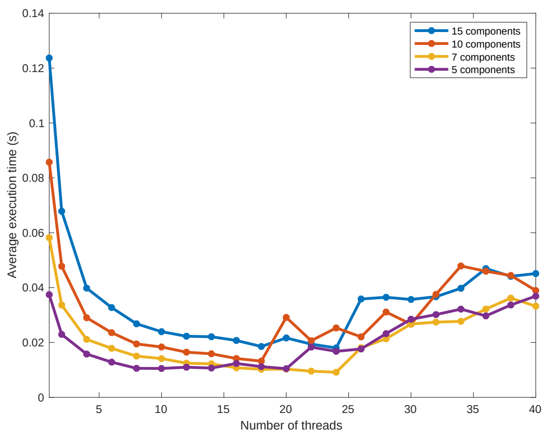
Scenario	RandomGreedy Avg. Exec. time for 500 iterations (s)			RandomGreedy Avg. Exec. time for 5000 iterations (s)		
	Python	C++ *serial*	C++ *parallel* (8 *threads*)	*Python*	C++ *serial*	C++ *parallel* (8 *threads*)
1	6.8489	0.0374	0.0106	69.2436	0.3667	0.0857
2	19.5444	0.0582	0.0150	200.9776	0.5769	0.1159
3	21.9022	0.0857	0.0194	239.1529	0.8374	0.1463
4	19.3566	0.1237	0.0268	241.41	1.2399	0.1901

Table 6.2: Scalability performance evaluation

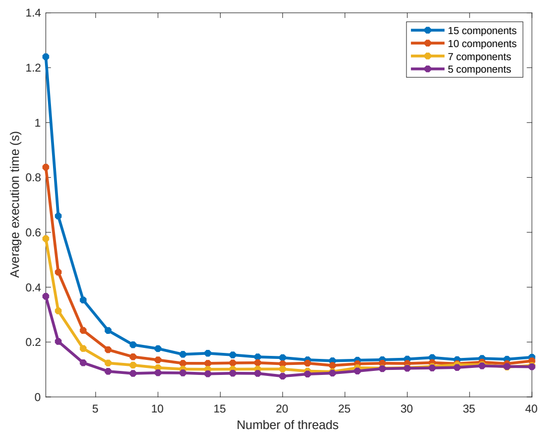
6.2. Parallel algorithm performance

To test the parallelized Random Greedy we exploited all the 20 cores of the Ubuntu server and we run the algorithm using up to 40 threads, with a step of 2. Again, we repeated the experiments with 500 and 5000 iterations and we considered the average execution time over 10 random instances of the four scenarios reported in Table 6.1.

Figure 6.3 shows the average Random Greedy running time as a function of the number of threads for each scenario.
In particular, from Figure 6.3a we can see that the duration of the experiments stops decreasing significantly at about 18 threads for scenarios 2, 3 and 4, and at 10 threads for the 5 components scenario. After the 18 threads threshold, we observe a strong and irregular increment of the execution times. This is probably due to the fact that 500 iterations are too little to make the best of the parallelization, and overheads start to dominate the total execution time.
In the 5000 iterations case (Figure 6.3b) instead, we have more regular results (validating the previous observation) and we reach the minimum execution time at about 24 threads for scenarios 2, 3, and 4, and at 20 threads for the 5-components case.



(a) 500 iterations.



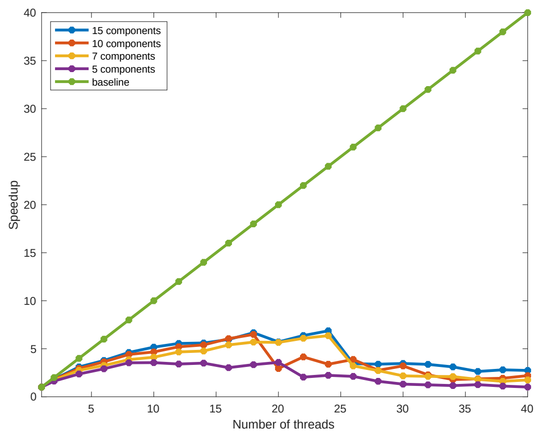
(b) 5000 iterations.

Figure 6.3: Parallelized RandomGreedy average execution time as a function of the number of threads.

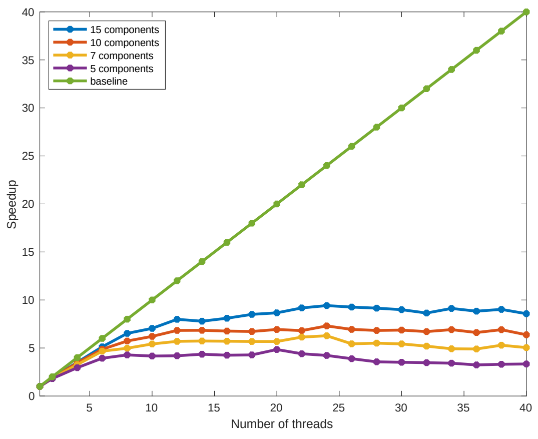
In Figure 6.4 we report the results of the speedup test for the exact same experiments mentioned above. Speedup is an index of how much a parallel algorithm is faster than the corresponding sequential algorithm and it is defined as following:

$$\text{Speedup}(n) = \frac{RunningTime(n)}{RunningTime(1)}$$

where n is the number of threads.



(a) 500 iterations.



(b) 5000 iterations.

Figure 6.4: Speedup of the parallelized RandomGreedy.

As one would imagine, the parallelization of the algorithm benefits best the large scale scenarios, with running times being at most 9 times faster than the serial version for the 15 components scenario with 5000 iterations, while for small scale the speedup barely reaches 5.

6.3. Validation of the Random Greedy

In this section, to further test the robustness of the Random Greedy, we exploit the heuristic method *HyperOpt* [12]. HyperOpt is an open-source `Python` library based on Bayesian optimization algorithms over awkward search spaces, which may include real-valued, discrete, and conditional dimensions.
In our analysis, we try to improve the best solution found by our algorithm providing to HyperOpt the N best solutions obtained by the Random Greedy. Therefore, in such a way, HyperOpt always obtains results as good as the Random Greedy.
This analysis is based on one of the ten random instances of the 5 components scenario (Table 6.1).
To qualitatively evaluate the different approaches, we define the percentage gain (denoted as *Cost ratio*) as follows:

$$\text{Cost ratio} = \frac{(HyperOptCost - RandomGreedyCost)}{RandomGreedyCost} \times 100.$$

The experiments have been run with λ ranging in [0.1, 1] req/s with step 0.01 req/s.

The cost comparison for 1000 and 5000 iterations is reported in Figure 6.5. We can observe that the HyperOpt never improves Random Greedy at 5000 iterations. For 1000 iterations it obtains small gains (lower than 2%). However, in practical applications, this would not be an issue since, as we proved in the previous sections, the execution time of the Random Greedy is so small that it can be safely run for 5000 iterations or more.

Nevertheless, HyperOpt is very slow (way slower than the `Python` implementation too): it took about three hours for 5000 iterations, for the heaviest configurations. Thus, using this optimization method is unpractical for real use cases.

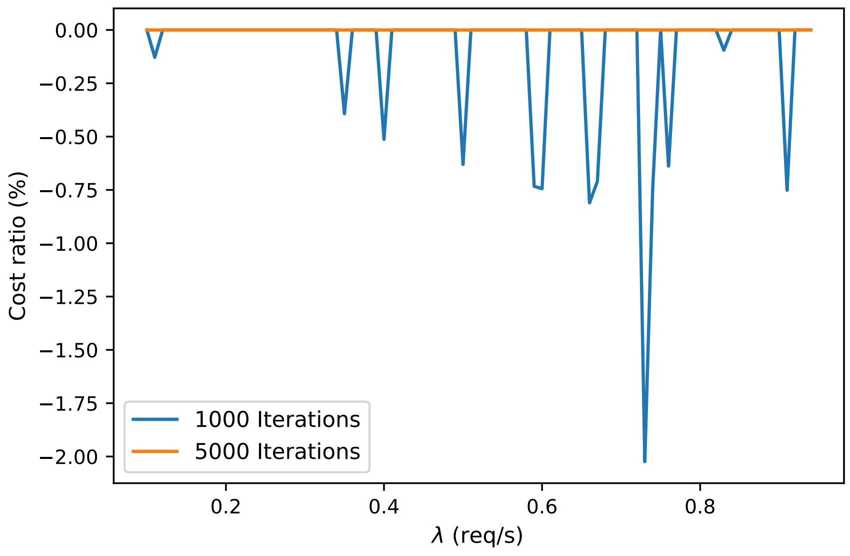


Figure 6.5: HyperOpt cost gain

7 | Conclusions and future developments

This work proposes a randomized greedy approach to support application component placement and resource selection in computing continua at design-time. The original package proposed in [2] has been improved exploiting the features of the faster `C++` and properly modifying many design patterns. Experimental results have shown that our implementation is approximately a hundred times faster than the original one, and the best solution costs are comparable in the two versions for all the considered scenarios. Moreover, parallelization can provide a huge speed-up, making the algorithm even more efficient.

As further developments, we are planning to introduce other meta-heuristics algorithms, such as the *Local Search* that can be used to tentatively improve the best solutions given by the Random Greedy, exploring their neighborhoods through atomic changes (e.g., component migration from Edge to Cloud or vice versa, deployments changes and scaling in or out VMs). However, Local Search may get stuck in local minima, thus *Tabu Search* will be implemented to escape them.

After that, we will introduce new performance models based on machine learning techniques. They are needed to improve the response times accuracy since, for some application, the performance models based on queuing theory or `pacsltk` [3] do not approximate the real responsiveness of the resources.

Finally, the library will be extended to manage run-time problems, which consists in adapting the initial design-time solution to a new one than can cope with the eventual workload variation. Mind that since this operation must be done online, it should be even faster than the resolution at design-time. Therefore, the huge reactivity improvement we achieved in this work will be key for the new run-time extension.

Bibliography

[1] AI-SPRINT: AI In Secure PRIvacy-preserving computing coNTinuum,, 2022. URL `https://www.ai-sprint-project.eu/`.

[2] Hamta Sedghani, Federica Filippini, and Danilo Ardagna. A Random Greedy based Design Time Tool for AI Applications Component Placement and Resource Selection in Computing Continua. In *2021 IEEE International Conference on Edge Computing (EDGE)*, pages 32–40, 2021. doi: 10.1109/EDGE53862.2021.00014.

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

[4] Matt Austern. Why you shouldn't use set (and what you should use instead), 2000. URL `http://lafstern.org/matt/col1.pdf`.

[5] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. Pybind11 — Seamless operability between C++11 and Python, 2016. URL `https://github.com/pybind/pybind11`.

[6] Niels Lohmann. JSON for Modern C++, 8 2022. URL `https://github.com/nlohmann`.

[7] Abhinav Ajitsaria. What Is the Python Global Interpreter Lock (GIL)?, 2018. URL `https://realpython.com/python-gil/`.

[8] geza. What are the disadvantages of nested vectors?, 2017. URL `https://stackoverflow.com/questions/45317057/what-are-the-disadvantages-of-nested-vectors`.

[9] Tobias Ribizel. Performance impact of nested vectors vs. contiguous arrays, 2017. URL `https://stackoverflow.com/questions/45747848/performance-impact-of-nested-vectors-vs-contiguous-arrays`.

[10] Peter Cordes. Using nested vectors vs a flatten vector wrapper, strange behaviour, 2017. URL `https://stackoverflow.com/questions/33093860/using-nested-vectors-vs-a-flatten-vector-wrapper-strange-behaviour`.

[11] Giulio Turetta. Come funzionano i web service, 2006. URL `https://www.html.it/pag/16449/come-funzionano-i-web-service`.

[12] HyperOpt GitHub. Hyperopt: Distributed Hyperparameter Optimization. `https://github.com/hyperopt/hyperopt`.

List of Figures

1.1 Directed acyclic graph for components 3
1.2 Example of component with its deployments 4

2.1 A use case of identifying wind turbines blade damage. 9
2.2 Candidate deployments of component C_4 11

5.1 Collaboration diagram for *AllResources* 28
5.2 Collaboration graph for *SystemData* class 29
5.3 Inheritance diagram for *Space4AI::BasePerformanceModel* 34
5.4 Collaboration diagram for *Space4AI::System* class 37
5.5 Collaboration graph of the class *Solution* 39

6.1 Average cost comparison 45
6.2 Relative cost increment 45
6.3 Parallel algorithm average execution times 47
6.4 Speedup of the parallel algorithm 47
6.5 HyperOpt cost gain 49

List of Tables

6.1 Scalability parameters 45
6.2 Scalability performance evaluation 46

List of Symbols

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
$\mathcal{L}^l$	Set of resource indices at layer l
n_j	Number of available devices of type $j \in \mathcal{J}_\mathcal{E} \cup \mathcal{J}_\mathcal{C}$
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 function $j \in \mathcal{J}_\mathcal{F}$
$d_{hj}^{i,hot}$	Hot request execution time of partition π_h^i on function $j \in \mathcal{J}_\mathcal{F}$

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 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 response time of a path P
Other parameters	
λ	Input exogenous workload
T	Time unit (one hour)
M	Constant for equilibrium conditions in Edge and Cloud
M_N	Constant for defining network delays

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
$\hat{y}_{hj}^i$	Number of Edge devices or VMs of type $j \in \mathcal{J}_{\mathcal{E}} \cup \mathcal{J}_{\mathcal{C}}$ assigned to partition π_h^i
$\bar{y}_j$	Maximum number of running VMs of type $j \in \mathcal{J}_{\mathcal{C}}$
$\tilde{w}_{h\xi}^{id}$	1 if partitions π_h^i and π_ξ^i are consecutive and deployed on different devices in the same network domain d
$\tilde{t}_{h\xi}^{id}$	Transfer time between π_h^i and π_ξ^i on network domain d
α_h^i	1 if π_h^i is the first partition of component i, 0 otherwise
w_h^i	1 if π_h^i is the last partition of component i, 0 otherwise
w^{ikd}	1 if i and k are consecutive components whose last and first partition, respectively, are executed on different devices in the same network domain d
$\tilde{R}_h^i$	Response time of partition π_h^i
R^i	Response time of component i
C	Cost of Edge devices or Cloud VMs
C_F, C_T	Execution cost and transition cost of FaaS configurations