
**5G ExPerimentation Infrastructure hosting Cloud-nativE Network Applications for public proTection and disaster RElief**

Innovation Action – ICT-41-2020 - 5G PPP – 5G

Innovations for verticals with third party services

# D2.4: 5G-EPICENTRE service placement

Delivery date: October 2023

Dissemination level: Public

| | |
|---|---|
| **Project Title:** | **5G-EPICENTRE - 5G ExPerimentation Infrastructure hosting Cloud-nativE Network Applications for public proTection and disaster RElief** |
| **Duration:** | **1 January 2021 – 31 December 2023** |
| **Project URL** | https://www.5gepicentre.eu/ |

This project has received funding from the European Union's Horizon 2020 Innovation Action programme under Grant Agreement No 101016521.

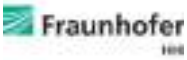
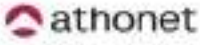
# Document Information

| Deliverable | D2.4: 5G EPICENTRE service placement |
|---|---|
| **Work Package** | WP2: Cloud-native 5G NFV |
| **Task(s)** | T2.3: VNF chain placement, re-routing and re-mapping |
| **Type** | Report |
| **Dissemination Level** | Public |
| **Due Date** | M34, October 31, 2023 |
| **Submission Date** | M34, October 31, 2023 |
| **Document Lead** | Fatemeh Tabatabaei (CTTC) |
| **Contributors** | Josep Mangues-Bafalluy (CTTC)<br>Manuel Requena (CTTC)<br>Hamzeh Khalili (CTTC)<br>Apostolos Siokis (IQU)<br>Almudena Díaz Zayas (UMA)<br>Jorge Márquez Ortega (UMA) |
| **Internal Review** | Apostolos Siokis (IQU)<br>Luis Cordeiro (ONE)<br>Carlos Marques (ALB)<br>Jorge Carapinha (ALB)<br>Ankur Gupta (HHI) |

# Document history

| Version | Date | Changes | Contributor(s) |
|---|---|---|---|
| V0.1 | 11/07/2023 | Initial deliverable structure | Fatemeh Tabatabaei (CTTC) |
| V0.5 | 28/08/2023 | 40% of the content | Apostolos Siokis (IQU)<br>Hamzeh Khalili (CTTC)<br>Manuel Requena (CTTC)<br>Almudena Díaz Zayas (UMA)<br>Jorge Márquez Ortega (UMA) |
| V0.8 | 19/10/2023 | 100% of content | Josep Mangues-Bafalluy (CTTC)<br>Fatemeh Tabatabaei (CTTC) |
| V0.9 | 25/10/2023 | Internal Review Version | Josep Mangues-Bafalluy (CTTC)<br>Fatemeh Tabatabaei (CTTC) |
| V1.0 | 27/10/2023 | 1st version with suggested revisions | Apostolos Siokis (IQU)<br>Luis Cordeiro (ONE)<br>Carlos Marques (ALB)<br>Jorge Carapinha (ALB)<br>Ankur Gupta (HHI) |
| V1.2 | 29/10/2023 | Revisions to document after 1st internal review | Josep Mangues-Bafalluy (CTTC)<br>Fatemeh Tabatabaei (CTTC) |
| V1.5 | 31/10/2023 | Final Version for Quality & Security Review | Konstantinos Apostolakis (FORTH) |
| V2.0 | 31/10/2023 | Final version for submission | Fatemeh Tabatabaei (CTTC) |

## Project Partners

| Logo | Partner | Country | Short name |
|---|---|---|---|
|  | AIRBUS DS SLC | France | **ADS** |
|  | NOVA TELECOMMUNICATIONS SINGLE MEMBER S.A. | Greece | **NOVA** |
|  | Altice Labs SA | Portugal | **ALB** |
|  | Fraunhofer-Gesellschaft zur Förderung der an-gewandten Forschung e.V. | Germany | **HHI** |
|  | Foundation for Research and Technology Hellas | Greece | **FORTH** |
|  | Universidad de Malaga | Spain | **UMA** |
|  | Centre Tecnològic de Telecomunicacions de Catalunya | Spain | **CTTC** |
|  | Istella SpA | Italy | **IST** |
|  | One Source Consultoria Informatica LDA | Portugal | **ONE** |
|  | Iquadrat Informatica SL | Spain | **IQU** |
|  | Nemergent Solutions S.L. | Spain | **NEM** |
|  | EBOS Technologies Limited | Cyprus | **EBOS** |
|  | Athonet SRL | Italy | **ATH** |
|  | RedZinc Services Limited | Ireland | **RZ** |
|  | OptoPrecision GmbH | Germany | **OPTO** |
|  | Youbiquo SRL | Italy | **YBQ** |
|  | ORamaVR SA | Switzerland | **ORAMA** |

# List of abbreviations

| Abbreviation | Definition |
|---|---|
| **3GPP** | 3rd Generation Partnership Project |
| **5G PPP** | 5G Public Private Partnership |
| **API** | Application Programming Interface |
| **CDF** | Cumulative Distribution Function |
| **CNF** | Cloud-Native Network Function |
| **CPU** | Central Processing Unit |
| **CRD** | Custom Resource Definition |
| **E2E** | End-to-End |
| **GA** | Grant Agreement |
| **(M)ILP** | (Mixed) Integer Linear Programming |
| **JSON** | JavaScript Object Notation |
| **K8s** | Kubernetes |
| **KPI** | Key Performance Indicator |
| **LCM** | Lifecycle Management |
| **LB** | Load Balancer |
| **MQTT** | Message Queuing Telemetry Transport |
| **NFV(O)** | Network Functions Virtualization (Orchestrator) |
| **OS** | Operating System |
| **PPDR** | Public Protection and Disaster Relief |
| **QoE** | Quality of Experience |
| **QoS** | Quality of Service |

| | |
|---|---|
| **RTT** | Round-Trip Time |
| **SDN** | Software Define Networking |
| **SF** | Service function |
| **SFC** | Service Function Chaining |
| **SoA** | service-oriented architecture |
| **TI** | Testbed Instance |
| **VNE** | Virtual Network Embedding |
| **VNF** | Virtual Network Function |
| **VPN** | Virtual Private Network |
| **VSF** | Virtualized Service Function |

# Executive summary

This deliverable constitutes the final report on the 5G-EPICENTRE service placement module. This module is part of the federation layer and is in charge of deciding the best cluster to deploy the service, where "best" is defined as the one contributing better to make an efficient (load balanced) use of computing resources, whilst fulfilling the stringent service requirements in terms of latency.

In this direction, this document first introduces the context of the problem, formulates it, and evaluates it under a variety of conditions, including varying loads and cluster capacities, number of clusters, service requirements, or other parameters, more related with the algorithm design itself, such as number of services per service batch (*i.e.*, number of services being deployed simultaneously in the emergency scenario).

This deliverable also presents the architecture built experimentally to integrate service placement itself, as well as the required auxiliary modules (*e.g.*, round-trip time measurement, data gathering).

The main contributions of this document are:

- To present a comprehensive solution to address the critical challenges associated with service management and resource allocation in emergency scenarios. This was achieved through the design and development of a built-in module, specifically for managing services based on their requirements in emergency situations. Our module ensures that services are deployed optimally, even under the stringent emergency conditions.
- To introduce a service placement algorithm, designed to propagate services across edge data centres, whilst ensuring optimal utilization of resources. This algorithm takes into consideration the dynamic and resource-constrained nature of edge computing environments, providing an approach to service deployment that maximizes the performance and availability of crucial services.
- To evaluate the proposed service placement algorithm based on integer linear programming, comparing it to a load balancing solution. It is shown that the former outperforms the latter in terms of resource usage efficiency in constrained edge computing clusters, since it allows handling higher CPU workloads and a higher number of services.
- To develop an innovative plugin that seamlessly integrates with the Karmada system, demonstrating the versatility of our approach. This plugin not only enhances the capabilities of the Karmada system, but also provides a framework that can be extended to integrate other modules and systems as needed.

# Table of Contents

# List of Figures

# List of Tables

# 1 Introduction

The 5G-EPICENTRE architecture consists of four main layers, namely front-end, back-end, federation, and infrastructure (refer to D1.4). The front-end layer is in charge of interacting with the user of the platform. The back-end layer receives the requests from the user (through the Portal), and manages the lifecycle of the corresponding experiments. Finally, the federation layer function is to aggregate the experimentation resources of the infrastructure layer (*i.e.*, the four testbeds of the project), and to expose them in a unified way to the back-end layer. Therefore, the back-end layer will be able to coordinate the lifecycle management of experiments in any of the 5G-EPICENTRE testbeds in the same way.

When deploying the services associated to a given experiment, one of the key functionalities is to select their location in the experimentation infrastructure. The focus of this deliverable is service placement, as a key functionality of the federation layer. When a service deployment request arrives to the federation layer, the service placement functionality selects the most appropriate cluster of the federation to deploy the service. This is also of interest in an emergency scenario, in which one could imagine multiple agencies arriving to the emergency spot, bringing their own resource-constrained computing equipment in the ambulances, firefighter trucks, police vans, *etc*. The computing capabilities of all these distributed resources could be aggregated by the federation layer, to be exposed to all agencies so that they use them as a single big cluster. In this case, service placement would be in charge of selecting the best service location, *i.e.*, the one that best contributes to making an efficient use of such constrained resources, while fulfilling the stringent requirements of Public Protection and Disaster Relief (PPDR) services. In general, this would mean trying to exploit the distant cloud for non-latency-constrained services (when possible), and do some sort of load balancing of the resources next to the emergency spot.

This document first gives the context of the service placement problem, formulates it along the lines explained above, and evaluates the proposed algorithm under a variety of conditions, including varying loads, number of clusters, and service requirements. Finally, the overall service-placement-related architecture that has been deployed in the 5G-EPICENTRE platform is also presented, along with the required auxiliary components, *e.g.*, for round-trip time (RTT) measurement, or for data gathering.

The following Sections describe the service placement designed and developed in the federation layer to consider the resources available at the network edge and across multiple geographically distributed testbed infrastructures in the context of 5G-EPICENTRE project. This deliverable is organized as follows: Section 2 deals with all aspects related with the design and evaluation of service placement. After that, Section 3 focuses on the practical implications of integrating service placement into the federation layer, and finally, in Section 4, the main conclusions are presented.

## 1.1 Mapping of project's outputs

The purpose of this section is to map 5G-EPICENTRE Grant Agreement (GA) commitments, both within the formal Task description, against the project's respective outputs and work performed.

Table 1: Adherence to 5G-EPICENTRE's GA Task Description

| 5G-EPICENTRE Task | Respective Document Chapters | Justification |
|---|---|---|
| T2.3: VNF chain placement, re-routing and re-mapping<br>*"[…] This Task will hence deal with optimal VNF chain placement, extended to consider the resources available at the network edge, and* | Section 3 – Experimental setup | This Section introduces the placement module, which is integrated into Karmada as an internal component. Karmada offers a federation environment that enables all testbeds to participate in, making |

| | | |
|---|---|---|
| *across the multiple geographically distributed testbed infrastructures, providing inter-connectivity among them".* | | them potential resources for deploying services. Within this context, we discuss the integration of the Integer Linear Programming (ILP) algorithm optimizer, and then optimizer with cluster resource plugin. |
| T2.3: VNF chain placement, re-routing and re-mapping<br>*"[…] The output of this Task will be an optimal VNF placement algorithm, that can: i) efficiently offload and redirect traffic between the Cloud and MEC resources available;"* | Section 2.2 – Mathematical model<br>Section 2.3 – Evaluation results | These Sections present the placement algorithm, which is implemented in the ILP solver. Optimal pod placement ensures that critical services are hosted closer to the edge, reducing latency and enhancing responsiveness for end-users. This, in turn, facilitates the offloading and redirection of traffic between resources. |
| T2.3: VNF chain placement, re-routing and re-mapping<br>*"[…] ii) offering flexibility in enabling the dynamic re-calculation of optimal placement to accommodate changing network dynamics and mobility support".* | Section 2.2 – Mathematical model<br>Section 3.1 – Architecture | These Sections present the approach for optimal location, or infrastructure, where a service should run. The scheduler, which is presented in Section 3.1, connects to other component of the Karmada system. It actively gathers real-time cluster conditions to make decisions that align with requirements and constraints, ensuring efficient resource allocation. |

# 2 Service placement

The advent of 5G has significantly broadened the possibilities in network communication, particularly in managing and deploying applications across diverse infrastructures. This is precisely what the federation layer in the 5G-EPICENTRE architecture aims at. However, an ongoing challenge lies in efficiently deploying services that demand cutting-edge communication technology to meet stringent requirements for low latency, high-speed, and high bandwidth. These requirements are crucial for providing effective decision-making support during emergency and disaster scenarios, where multiple agencies deploy their resources in the emergency areas in a PPDR context [1]. As multiple PPDR agencies rush to deploy their resources in an emergency zone, the service orchestrator (by offering a holistic view of the available resources, current deployments, and on-ground requirements), ensures that each agency's effort is channelled in the right direction. It aids in avoiding duplication of efforts and ensures that resources, especially the scarce, or critical ones, are utilized where they can have the most significant impact. The importance of the federation layer doesn't end at resource coordination. Another pivotal aspect in PPDR scenarios is the service placement, which ensures that services are deployed based on service requirements with a focus on a proactive approach that ensures effective resource utilization from the outset.

Service placement is the strategic allocation of computational and network resources to ensure that services, especially those that are latency-sensitive, operate efficiently and effectively. This is particularly vital in PPDR scenarios where quick decisions are made based on the real-time infrastructure data. A delay, even of a few milliseconds, in providing service to the end user, can lead to missed opportunities in disaster relief, inefficient resource allocation, or, at worst, loss of life.

The importance of KPI-based service placement in PPDR scenarios can be understood from following broad perspectives:

- **Latency constraints:** Many PPDR services, such as real-time video feeds from drones, coordination among first responders, or health telemetry from injured individuals, demand low-latency operations. In such scenarios, services need to be placed close to where the action is, often on the edge of the network or on-site, to ensure timely data delivery. These services do not have the luxury of time that many conventional applications might have. In essence, in a disaster relief context, latency isn't just a technical metric; it has palpable human implications**.**
- **Resource constraints**: Emergency zones are typically characterized by the unpredictability and scarcity of resources. There might be limited bandwidth due to damaged infrastructure, fewer computational resources because of power outages, or compromised network integrity due to environmental challenges. This scarcity underscores the need for smart service placement. Resources need to be utilized judiciously, ensuring that the most critical services get precedence. Moreover, as different agencies deploy their assets, there is a pressing need for coordination to prevent resource contention and ensure that every piece of technology deployed works in concert to achieve the common goal of relief and protection.

In conclusion, the challenges presented by PPDR scenarios evolve. Natural disasters, public emergencies, and other large-scale events test the limits of our technological frameworks. Central to navigating these challenges is the science and strategy of service placement. It is a domain where technology meets strategy, and where efficient algorithms can have real-world humanitarian impacts. As we delve further into this deliverable, we will explore the nuances, methodologies, and innovations that define Key Performance Indicator (KPI)-based service placement in PPDR scenarios, highlighting its significance in contemporary disaster relief and public protection paradigms.

## 2.1 State-of-the-art on service placement

The service placement problem refers to the challenge of determining the optimal placement of a virtual network service within a multi-cluster network infrastructure. In the context of network architecture, VNFs serve as software-based network services, often referred to as Cloud-Native Network Functions (CNFs) in experimental settings. These services encompass functionalities like firewalls, load balancers, routers, and deep packet inspection. There has been a lot of work on virtual network function (VNF) placement during recent years. This is why we refer to this work as a related source of inspiration, due to: (i) the similarity of the placement problem and the network and service parameters involved; and (ii) the potential applicability of the same kind of techniques and ideas.

When a network service is composed of multiple VNFs that need to be traversed in a specific order, it forms a VNF chain. In general, the placement problem arises when deciding on the appropriate locations within the network, where each VNF of the chain should be deployed to meet certain objectives, or constraints. The requirement of services can encompass multiple objectives, such as reducing cost, minimizing the end-to-end latency, reducing energy consumption, ensuring reliability, *etc*. However, the trade-offs between these objectives can lead to several conflicting issues, as placing several functions in the same device can cause scalability problems.

There is a sizeable load of service-oriented architecture (SoA) works published for service placement in a single domain [2]–[7], and works that tackle both orchestration and lifecycle management (LCM) techniques in an Network Functions Virtualization (NFV)-based multi-domain environment, where multiple operators cooperate, in order to provide the 5G vertical services and applications [8]-[10]. From the aforementioned works, [7] [8] are tackling the issue from a cost-aware scope, when it comes to edge-core environments, where pricing policies and resource availability may greatly vary.

Regarding the single domain SoA works, authors in [2] propose a resource allocation principle for energy-aware service function chain (SFC) for software define networking (SDN)-based networks. Multiple heuristics are explained for different optimization problems. The authors consider a single administrative domain with a centralized datacentre. The VNF/SFC placement formulation types are provided using integer linear programming (ILP) formulation in [2] [4] [6]; mixed integer linear programming (MILP) formulation in [3] [7]; and virtual network embedding (VNE) formulation in [5]. Additionally, [2]–[4] [6] [7] provide heuristic-enabled placement algorithms, while [4] provides approximation placement algorithms. Considering the placement method, [2] [3] [6] are focusing on the Quality of Service (QoS), while [4] on the cost. Authors in [5] [7] provide both a cost and QoS–aware placement method. The work described in [6] is leveraging the leafspine network topology (data centre network topology that consists of two switching layers, a spine and a leaf), while in terms of NFV Orchestrator (NFVO) capabilities, [4] provides LCM functionalities. With respect to the architecture, none of the aforementioned works use decentralized resources (*i.e.*, edge), they use only centralized datacentre environments (*i.e.*, core). Finally, with reference to the experimental results, all the aforementioned works provide simulation-based results, while none of them provides a testbed environment for further verification.

Taking a deeper look into the SoA works, authors in [11] discuss the challenge of latency in 5G network scenarios, emphasizing its significance in mission-critical environments, where delays are highly sensitive. To address this issue, the authors propose optimizing the service infrastructure placement to minimize delays in the service access layer. The placement problem in a Fog Computing/ NFV environment is mathematically considered as a MILP problem. In [7], the authors addressed challenges in 5G networks, focusing on reducing end-to-end latency by optimizing SFC placement and resource allocation at the network edge. They propose a comprehensive MILP model that encompasses user association, SFC placement, and resource allocation in a 5G network infrastructure. The problem formulated aims to minimize end-to-end latency, service provisioning cost, and the impact of virtualized service function (VSF) migrations on user experience.

Authors in [9] provide a 5G operating system (OS), in order to lower the complexity of the underlying 5G infrastructure in a multi-domain environment. Their work is limited, to a high-level architecture of the 5G OS, and the

placement, orchestration or LCM techniques are not explained in depth. Finally, authors in [10] provide a multi-cloud orchestration solution that is completely decentralized. They are using ILP in order to formulate the problem, while they provide three different optimal-based solutions for the VNF placement problem, focusing on cost, quality of experience (QoE) and the game theory-based trade-off between the cost and QoE, respectively. This work does not support migration functionalities, and the simulation-based results are very limited.

## 2.2 Mathematical model

In this Section, we present a mathematical model, designed to address the service placement problem in 5G networks. The focus is on the analysis of the efficiency of resource usage at the edge, among those clusters that fulfil the latency requirements. This is precisely where resources are scarcer, since computing devices hosting the PPDR services are carried in emergency vehicles. The ubiquity and criticality of 5G networks underscore the importance of efficiently placing services in the network, thereby ensuring optimal performance and resource utilization. Our model focuses on the objective of minimizing the maximum load of clusters as resource optimization, which in the end, results in a more efficient load balancing. The formulation is structured to capture the nuances of both the service and infrastructure layers, providing a comprehensive approach to the problem at hand.

Let the set of services be $s \in S$. Each service s has specific characteristics and requirements. The $cpus$ represents the central processing unit (CPU) demand of service $s$. This is the amount of computational resources required by the service to function optimally. $ls$ represents the requested latency for service $s$. It is an indication of the maximum acceptable latency for the service to deliver its function to the end users. Each cluster in the infrastructure layer can be characterized by $c \in C$, where $C$ denotes all clusters in the federation. Each cluster $c$ has its own attributes, as $cpuc$ represents the aggregated remaining available CPU capacity of cluster $c$. It denotes the computational resources that are currently free, and can be allocated to services. $lc$ represents the latency from the master node of cluster $c$ to user equipment (UE). It provides an understanding of the time taken for a data packet to travel between the node and the end user. The core objective is to minimize the maximum CPU utilization across all clusters. To encapsulate the deployment of service $s$ on cluster $c$, we introduce a binary decision variable $x_{s,c}$ defined as:

$$\begin{cases} 1 & \text{if service } s \text{ is deployed in cluster } c \\ 0 & \text{otherwise} \end{cases}$$

Now let us capture the objective function as:

$$min\left(max_{c \in C}\left(\sum_{s \in S} cpu_s \times x_{s,c}\right)\right) \quad (1)$$

Now let us present the constraints as follows:

$$\forall\, c \in C: \sum_{s \in S} cpu_s \times x_{s,c} \leq cpu_c \quad (2)$$

Equation (2) shows that the aggregated CPU utilization by the services deployed on a cluster should not exceed the cluster's CPU capacity.

$$\forall\, s \in S: \sum_{c \in C} x_{s,c} = 1 \quad (3)$$

Equation (3) highlights that the service can only be deployed non-fractionally.

$$\forall\, s \in S, \forall\, c \in C: l_s \times x_{s,c} \leq l_c \quad (4)$$

Equation (4) denotes that the latency of the deployed service should not exceed the latency from the cluster to the user. In summary, this mathematical model offers a robust framework to address the service placement problem in 5G networks. By optimizing for both CPU utilization and latency, we ensure that services are placed in a manner that maximizes the overall efficiency and performance of the network.

## 2.3 Evaluation results

To validate the proposed optimization model, we conducted extensive simulations comparing our approach with the Load Balancer (LB) benchmark algorithms. The strategy employed by LBs is the highest CPU available method. When a new request comes in, the LB allocates it to the server with the highest available CPU capacity at that moment. In both strategies, if no cluster has the required capacity, the task may either be queued, until a server becomes available, or a new server might be provisioned, depending on the context.

The simulation environment emulates a typical 5G network, comprising four clusters with a fixed capacity of 16 cores (or 16000 millicores) each. The computing requirements of the PPDR services to be deployed over the platform exhibit varying CPU demands, ranging from 2000 to 7000 millicores each (See Table 2 for details). The range of demands of the service can be changed uniformly, which represents a realistic and dynamic network environment, where service demands fluctuate. The total number of service requests for each repetition is fixed to 15. For the optimization tasks within our simulations, we have used the IBM ILOG CPLEX ILP solver [12]. The presented method was validated, by comparing the performance metrics with the benchmark.

Table 2 presents the main parameters of the scenario, and the values considered. Other scenario parameters were evaluated with similar conclusions obtained. Therefore, the scenario parameters selected below, and the associated discussion for each of the graphs, is equivalent to the one generated with different variations of these parameters. In this sense, this scenario is taken as a representative one, to show the typical performance of ILP compared to LB for the scenarios of interest.

Table 2: Scenario parameters

| Name | Range of values | Explanation |
|---|---|---|
| Number of services | 15 | Number of services that are deployed in a given simulation repetition. In the case of ILP, they are handled in batches/groups. |
| Number of loads | 5 | Number of loads simulated. Selected to vary from situations where all services easily fit the available resources, to situations in which the services do not fit, and some cannot be allocated. |
| Service load range (millicores) | [2000 .. 7000] | Increasing load per service is generated by increasing the rightmost range in 1000-millicore steps, starting at 3000 and ending at 5000. The service load for each service is randomly generated in the corresponding range, following a uniform distribution. |
| Number of clusters | 4 | Representing the available computing resources in each of the emergency vehicles (*e.g.*, ambulance, firefighter's vehicle, police van). |
| Cluster capacities (millicores) | 16000 | Based the specifications of some products specifically designed to be used in emergency scenarios. |

| | | |
|---|---|---|
| Number of repetitions | 10 | Different random service loads are generated in each repetition. |

Service loads are selected so that they fulfil the following requirements:

1. When randomly generated, they are uniformly distributed in the range selected for a given load. For instance, for the first load the range is [2000 .. 3000], for the second one it is [2000 .. 4000], *etc*. (see Table 3).
2. They allow evaluating the system in a variety of situations that range from unloaded (where no cluster reaches its maximum capacity, and so, services are easily placed), to fully loaded (where all services cannot be placed due to lack of resources). It is precisely under these conditions, where the resource efficiency of the algorithm comes into play.

Table 3: Loads generated

| Min | Max | Average | Total average (all services) | Per cluster average workload | Normalized cluster load |
|---|---|---|---|---|---|
| 2000 | 3000 | 2500 | 37500 | 9375 | 0,5859375 |
| 2000 | 4000 | 3000 | 45000 | 11250 | 0,703125 |
| 2000 | 5000 | 3500 | 52500 | 13125 | 0,8203125 |
| 2000 | 6000 | 4000 | 60000 | 15000 | 0,9375 |
| 2000 | 7000 | 4500 | 67500 | 16875 | 1,0546875 |

Table 3 presents the characteristics of the loads generated in the simulations for the five loads, including the range (characterized by min and max) from which service workloads are randomly selected uniformly. The third column presents the average load around which the services should be selected, as can be seen in the Figures below, when presenting the average service load in the X axis. The fourth column presents the total load that would be generated by all services in each repetition (assuming that each service had a workload around the average one). Finally, the two last columns provide an estimate of the average load that each cluster would have to absorb, assuming that the total workload could be exactly evenly distributed, which is not the case due to the services having varying workloads that must be placed as a single unit. Since the cluster capacity is 16000 millicores, the normalized cluster load shows that, for the last loads evaluated, there will be services that for sure will not fit in the system, since we are requesting more than the total system capacity. When due to the randomness of the service workloads generated, the services do not fit in the total system capacity, services are randomly discarded. It is important to notice that the service workloads are generated in each repetition based on the approach explained before.

The performance of the algorithms is evaluated based on how well the scarce resource of the edge clusters are used. Other related metrics allowing to characterize how resources are allocated are also obtained. The selected metrics are presented in Table 4.

Table 4: Evaluated metrics

| Name | Explanation |
|---|---|
| Not allocated CPU workload | Amount of CPU workload coming from services that did not fit in the system. In some cases, this is due to the requesting more total workload than the sum of the capacities of all clusters. In this case, services are randomly discarded for all algorithms evaluated. The key aspect to highlight is that the difference in this value among algorithms is due to the efficiency in service allocation. |
| Not allocated services | Same as above parameters, but in terms of number of services affected. The *not allocated* CPU workload (above metric) may come from many services with small CPU workload request, or from a few services representing much bigger demands. |
| Maximum cluster load | Load of the cluster with the highest load among all those available in the system. This parameter is interesting because it shows how balanced the workload allocation is among clusters. |
| Cluster load | Related with the previous metric, a deeper analysis of not just the maximum load, but the general behaviour in terms of workload allocation for all clusters (*e.g.*, averages, cumulative distribution functions - CDFs) helps in understanding how each algorithm performs. |
| Variability of cluster load | Quantifying the variability of load allocation in each cluster measures how evenly balanced the workload distribution is. |

### 2.3.1 Analysis of *not allocated CPU workload*

Figure 1 presents a comparison of ILP *vs.* LB in terms of average *not allocated CPU workload* over all repetitions of a given service load. There are 5 points per curve, corresponding to the 5 loads evaluated (Table 4). The difference in performance comes from the better use of resources by the ILP algorithm, which, for all repetitions is capable of making all services fit in the resources, as long as the total CPU workload demand is below the total capacity of the system (*i.e.*, the sum of all cluster capacities). If in one repetition, the total CPU workload requested (*i.e.*, the sum of all workloads of all services of that repetition) is above system capacity, services are randomly removed from the service request queue for both ILP and LB (*i.e.*, exactly the same service requests are removed in both cases). Therefore, ILP shows a much better performance, precisely when it is more needed, that is, when operating close to system capacity. This is particularly relevant in a system with scarce edge resources, such as that of an emergency scenario.

To further assess the observed behaviour and more specific cases, Figure 2 presents the *not allocated CPU workloads* for all the repetitions of the simulations. The X-axis represents the average service workload for all the service requests of a given repetition (*i.e.*, the average over the 15 services being deployed in that repetition). In this case, given the randomly generated CPU workload requests of each service, the points are scattered through all the ranges of load values. The same trend observed in the previous Figure is also observed here, since, when comparing ILP *vs.* LB for a specific X-axis value, ILP is always below LB, hence ILP always allows allocating more CPU workload, no matter the combination of random service requests. Notice that for those points, where the light grey points are not represented, it is because they overlap with LB. In this case, this happens in the leftmost part of the graph, for which the *not allocated CPU workload* is 0, due to the system capacity being much higher than the total demand.

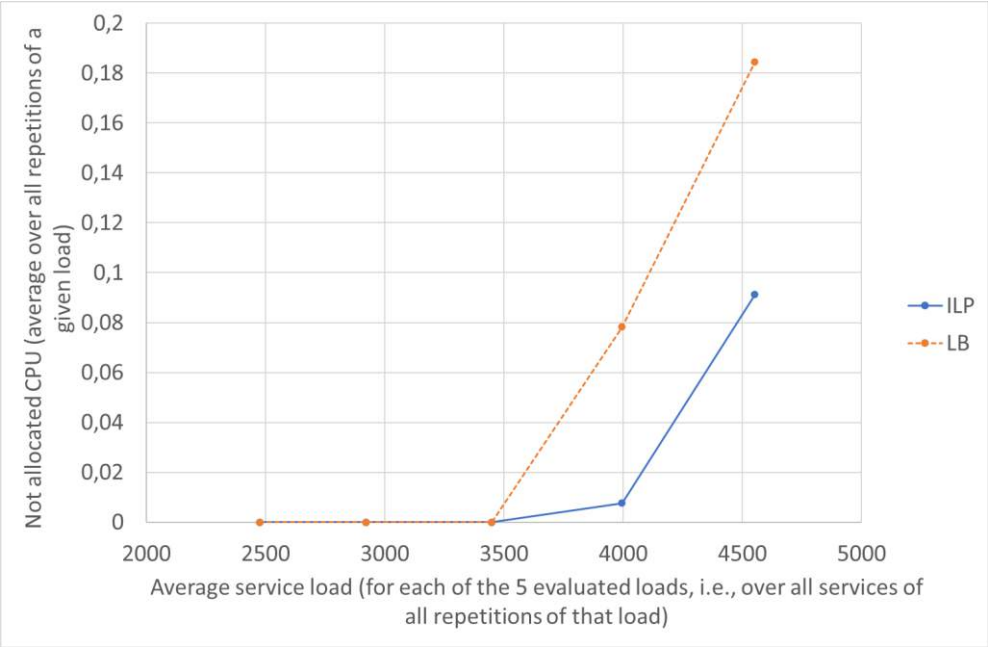



Figure 1: Comparison of ILP *vs.* LB: Average *not allocated CPU workload* (over all repetitions of a given service load).

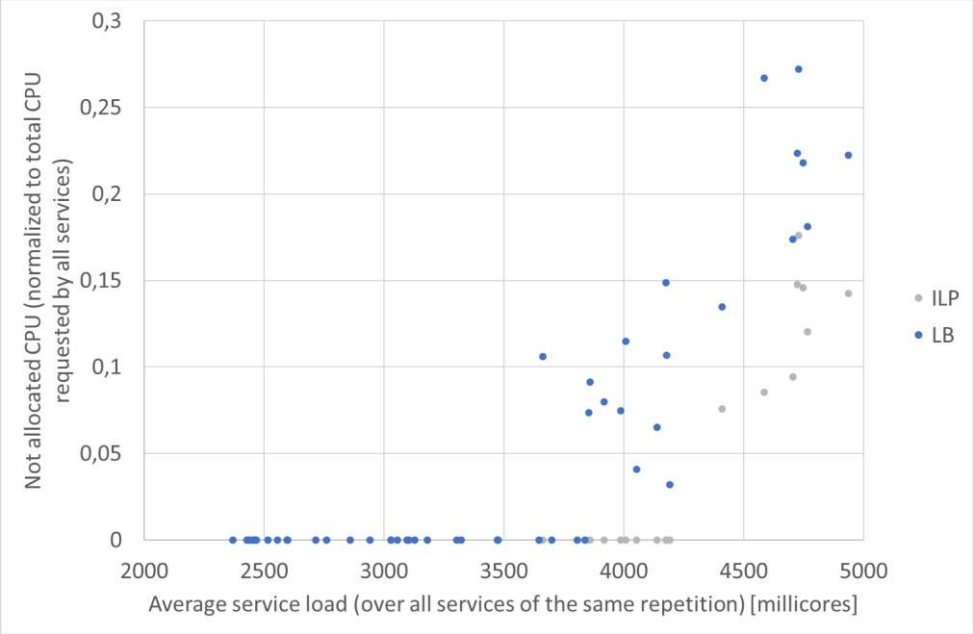



Figure 2: Comparison of ILP *vs.* LB: *Not allocated CPU workload* normalized to the total CPU workload requested by all services of a given repetition. One value per repetition over all loads evaluated.

Another characteristic that can be observed, and that will be further analysed in the Sections below, is that the dispersion of points in LB is broader. This is a consequence of a not-so-optimal use of resources in the clusters, which generate losses of new services arriving with big CPU workloads, that do not fit in any cluster, because the previously deployed services were not packed as evenly as in ILP. In the following subsections, we observe and discuss in more detail this characteristic.

### 2.3.2 Analysis of *not allocated services*

Figure 3 presents the same analysis as above, but this time focusing on the average number of services lost due to not fitting the system. As mentioned above, this only happens in ILP because the total requested workload is above system capacity. On the other hand, it is common to lose services in LB, even when the total workload is well below the system capacity, due to the way LB takes placement decisions, which focuses on instantaneous system conditions, as opposed to a more global approach for all services, as done by ILP. The Y-axis represents the average over all repetitions of a given load.

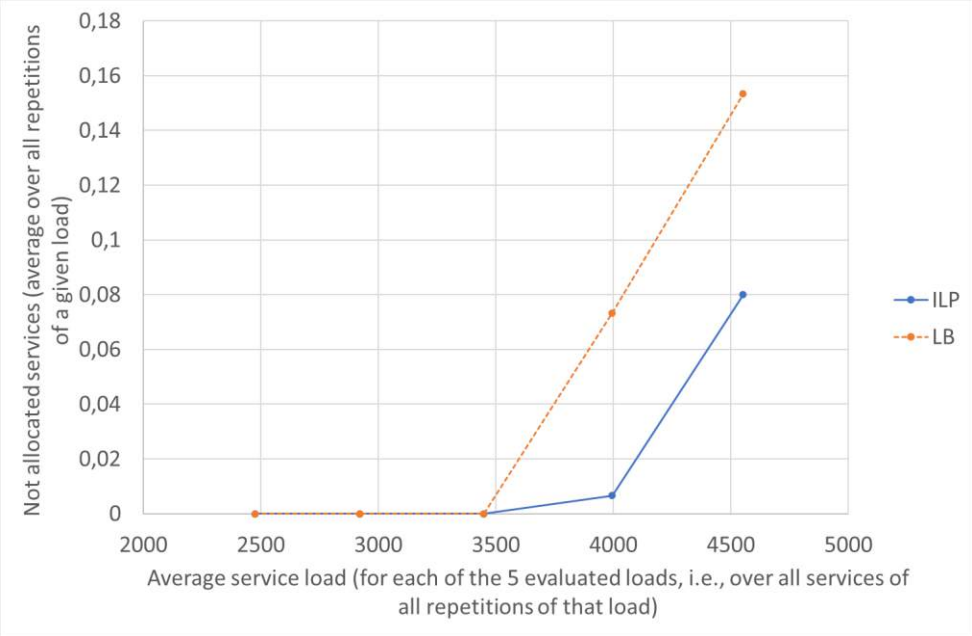



Figure 3: Comparison of ILP *vs.* LB: Average *not allocated services* (over all repetitions of a given service load).

When analysing all repetitions (Figure 4), despite the fact that the *not allocated services* is a natural number, and so the output presents discrete values, similar conclusions as above can be drawn, *i.e.*, for a fixed X-axis value, ILP always loses the same (if 0 services are lost in unloaded systems), or a lower number of services (when close to system capacity) than LB.

### 2.3.3 Analysis of load distribution in clusters: *maximum cluster load*

Figure 5 represents the load of the cluster (normalized to cluster capacity), that has maximum load among all clusters for a given repetition. That is, at the end of the repetition, when all services are deployed (or when they are not deployed, because they do not fit in the system), we examine the CPU being used in each cluster, and that with a higher value is presented in the graph. Notice that in each repetition, a different cluster may be chosen, depending on how the random service requests were deployed by each algorithm. Since the goal of the ILP algorithm is to minimize such maximum value, one would expect ILP to always be below LB. This is indeed the case for most repetitions, as observed in the Figure, since for a given X-axis value, the circle is always below the diamond. However, one can also observe average service loads for which this is not the case. And this behaviour is much more common, as we get closer to the highest evaluated loads. The explanation is that since ILP is capable of deploying more services, more CPU workload is deployed in the clusters of the system, and so, in general, there may be clusters that have a maximum CPU usage, bigger than the maximum of LB.

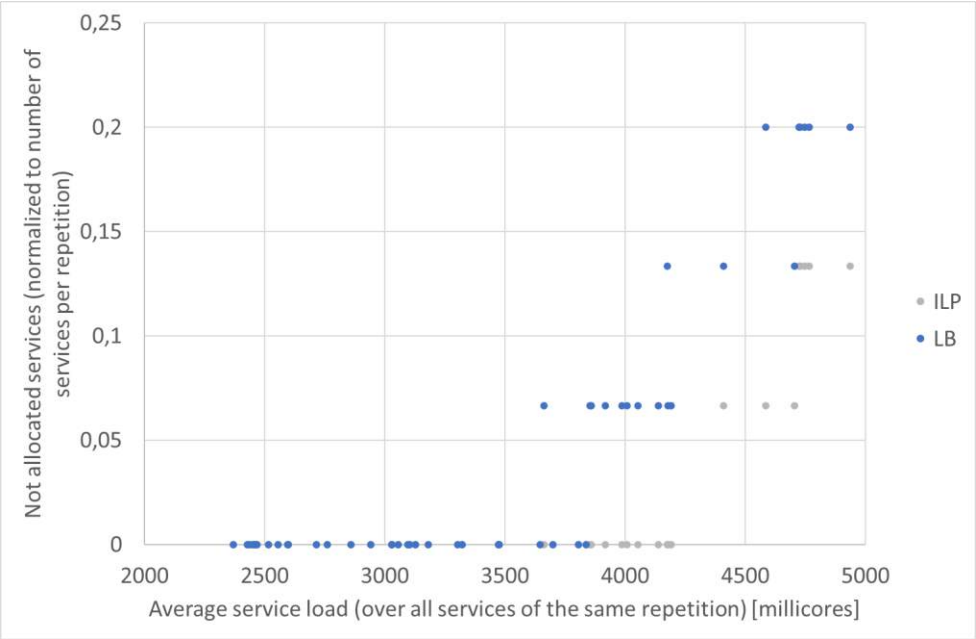



Figure 4: Comparison of ILP *vs.* LB: *Not allocated services* normalized to the number services of requested in each repetition. One value per repetition over all loads evaluated.

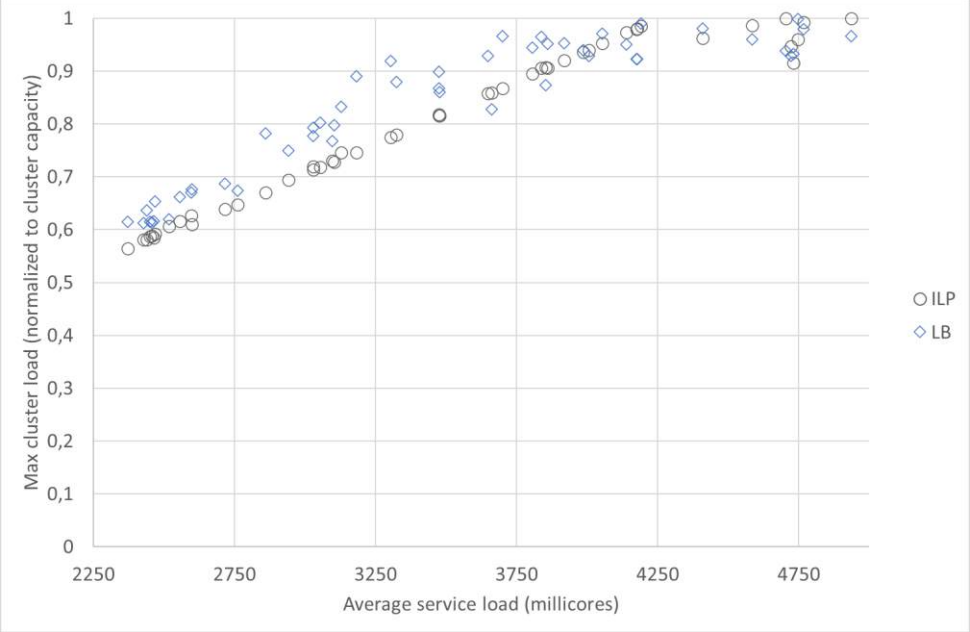



Figure 5: Comparison of ILP *vs.* LB: *Maximum cluster load* normalized to cluster capacity. One value per repetition over all loads evaluated, *i.e.*, in each repetition, the point represents the normalized load of the cluster with highest load among all clusters.

Another observation is that, again, the LB shows a broader range of values than ILP, for which trend follows a quite well-defined linear trend. This also shows that the load in the clusters is more evenly balanced for ILP, since even if the cluster in a given repetition with the maximum load is different from that of another, the difference is not so big, since the dispersion seems to be much smaller, and so, the other clusters are expected to have very similar CPU usage. This is confirmed in the following Sections.

### 2.3.4 Analysis of load distribution in clusters: *per-cluster load*

Figure 6 shows more details of the behaviour of each of the clusters for both schemes (asterisks for LB and dots for ILP). Again, the same behaviour explained above is confirmed, for which ILP seems to load balance evenly, since all dots are grouped in an almost perfect line. However, for low and high loads, this is not the case. In both cases, this is due to the varying workloads of services. Therefore, despite being evenly distributed, if one bigger service is deployed in a given cluster, and those of the other can be more easily packed, we will notice a difference between cluster loads due to this randomness of service workloads. For higher loads, this randomness may result in a big service not being deployed, because it does not fit in the system, and so the total workload to be deployed would be much lower than other repetitions, where there are no such big differences among services. In any case, the dispersion in LB is much higher, as also observed before, which results in some edge cluster being quite full, while others have much lower occupancy. This would allow the latter clusters to receive more services, but if it is precisely in the more loaded ones, where it should be deployed, due to service requirements. This would pose some problems in emergency scenarios. This situation is less likely to happen with ILP.

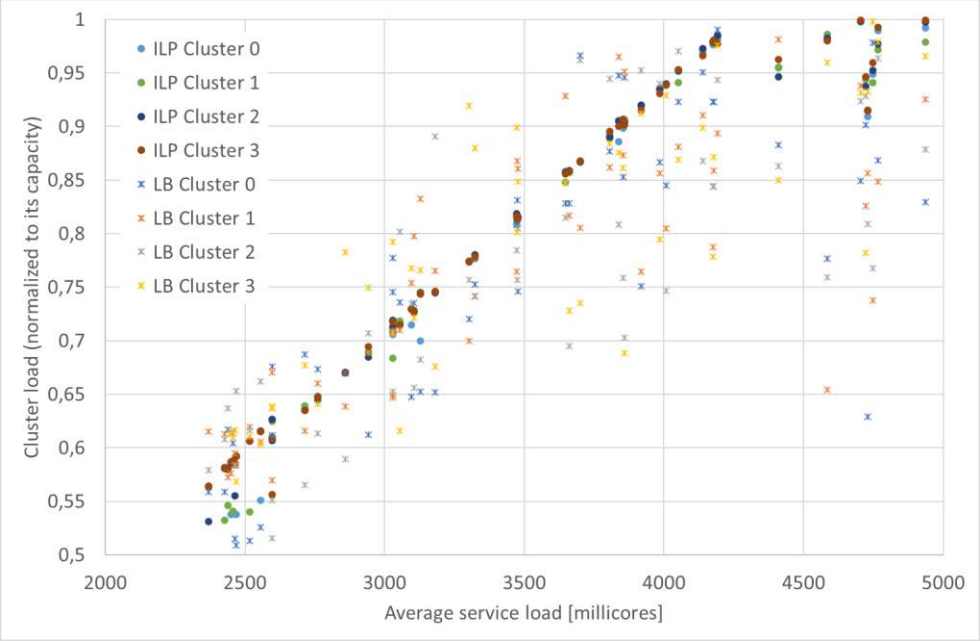



Figure 6: Comparison of ILP *vs.* LB: *Cluster load* of each cluster normalized to cluster capacity. One value per repetition over all loads evaluated, *i.e.*, in each repetition, the point represents the normalized load of each cluster at the end of each repetition. There is one series of values for each of the four clusters for each algorithm evaluated.

Figure 7 presents the cumulative distribution function of the normalized *cluster load* for each cluster. We are particularly interested in the behaviour close to cluster capacity, since it is in these regions where the resource efficiency of algorithms comes more into play. In this region, the cumulative distribution function (CDF) graph shows that ILP has lower normalized cluster loads than LB for all clusters, which implies that there is more room to receive more services, hence to host more CPU workloads. As deduced from the graph, it is more common to have clusters operating in the mid loads region (from 0.5 to 0.75 normalized loads). As previously discussed, this is due to the global load balancing capabilities of ILP, because it considers all the service requests and all the available resources in a global manner. This allows distributing the services more evenly, resulting in higher resource consumption efficiency.

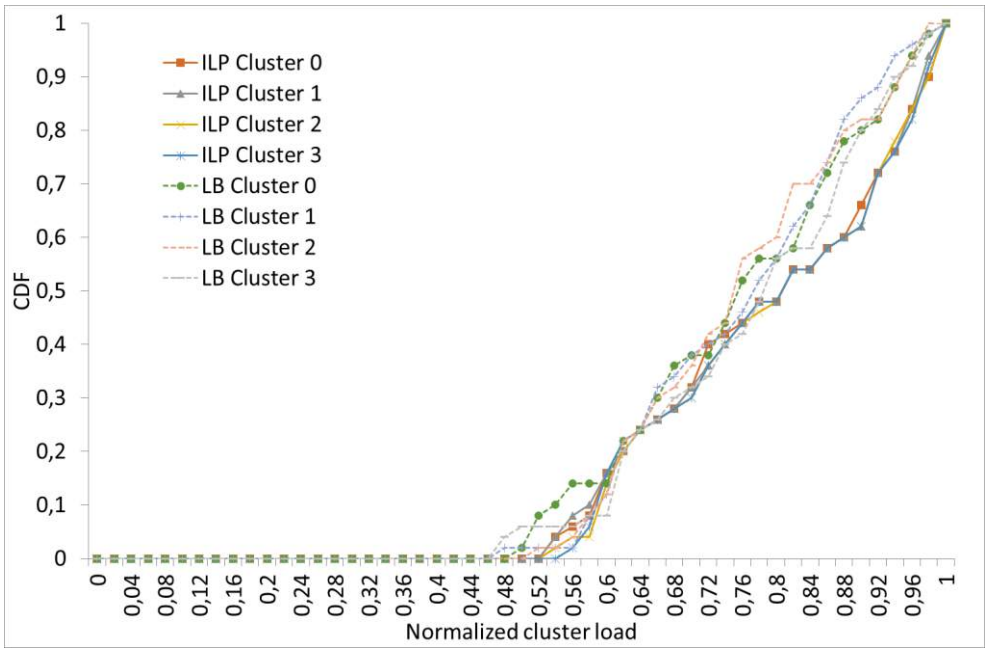



Figure 7: Comparison of ILP *vs.* LB: Cumulative distribution function of each *cluster load*, normalized to cluster capacity. There is one series of values for each of the four clusters for each algorithm evaluated.

Another way of representing this behaviour is by averaging over all repetitions of a given load. In this case, the trends are clearer, since all clusters behave in the same way in ILP, and there is more dispersion in LB. In fact, higher workloads can be absorbed by ILP, which can deploy services to almost completely fill the system. This is not the case for LB, whose average cluster load is substantially smaller at high average service loads.

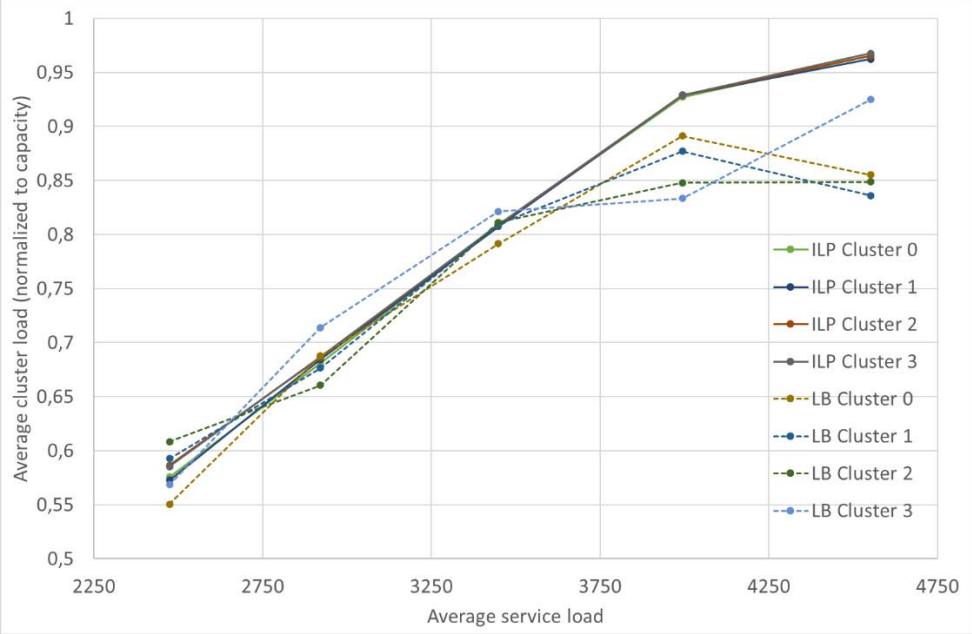



Figure 8: Comparison of ILP *vs.* LB: Average *cluster load* normalized to cluster capacity over all repetitions of a given load. There is one series of values for each of the four clusters for each algorithm evaluated.

Finally, Figure 9 gives another point of view of the same aspect, *i.e.*, a more homogeneous behaviour of clusters when using ILP and a more even distribution of loads. This is observed in the overlapping standard deviation curves for ILP, and the higher dispersion of values and higher values for LB.

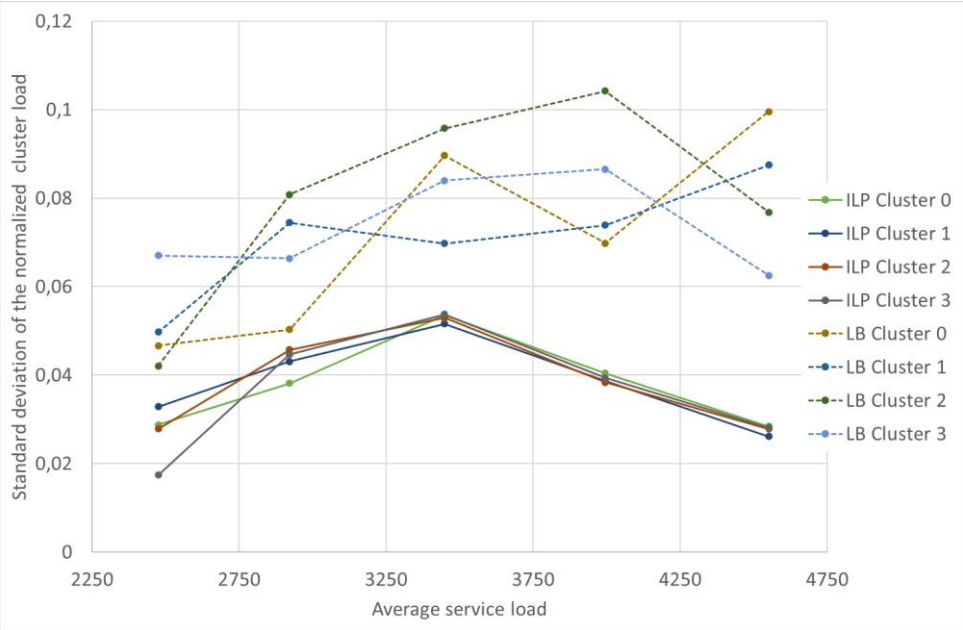



Figure 9: Comparison of ILP *vs.* LB: Standard deviation of *cluster load*, normalized to cluster capacity over all repetitions of a given load. There is one series of values for each of the four clusters for each algorithm evaluated.

# 3 Experimental setup

To ensure a comprehensive understanding, this Section will delve deeper into the software elements utilized. We will discuss the specifications of the cross-testbed federation, elucidate the process by which various Kubernets (K8s) clusters join the federation, and, most importantly, unpack the logic and functioning of the service placement. Our experimental approach centres around leveraging the capabilities of *Karmada* (Open, Multi-Cloud, Multi-Cluster Kubernetes Orchestration)[1], an open-source platform designed to deploy application containers across multi cluster environment, to harness the advantages of an orchestrator, where multiple K8s clusters join to form an integrated, synchronized environment. This configuration facilitates a centralized control – attributes essential for agile and responsive PPDR solutions. The idea of cross-testbed federation is conceptualized in Figure 10. The K8s-based orchestrator manages and provides access to service resources for the individual testbed containerized workloads, hence providing multi-clustered K8s orchestration across different domains, which allows testbed federation and synchronization.

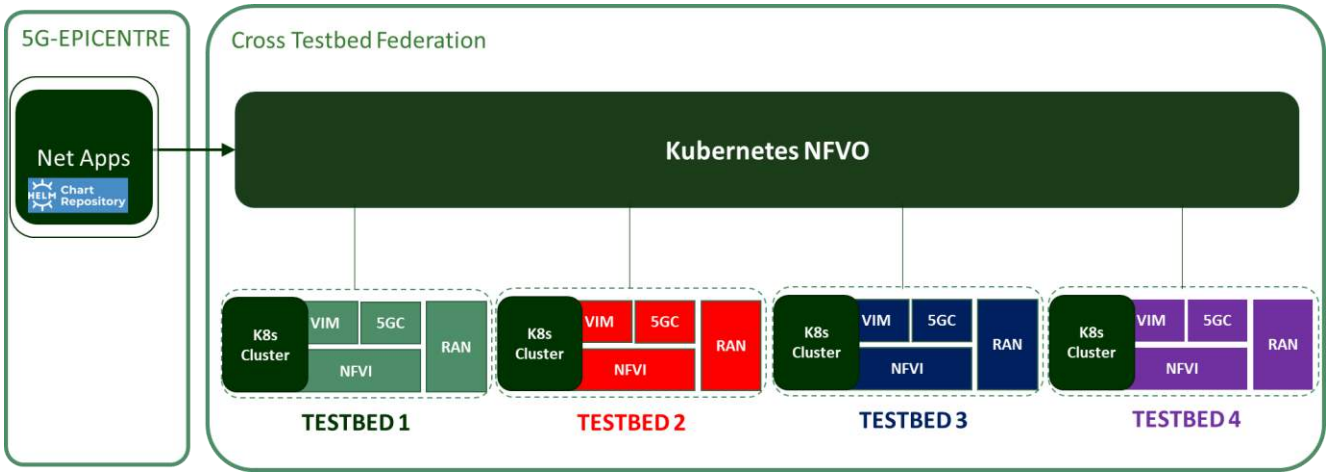



Figure 10: Cross Testbed Federation for 5G EPICENTRE.

Using Karmada, the K8s management system enables cloud-native applications to run across multiple K8s clusters, with no changes to the application. It allows application deployment and resource management on multiple clusters, known as *member clusters*, from the centralized Karmada control plane. Moreover, Karmada allows the federation of any K8s resources to be used in a multi-cluster environment. As an initial approach, Karmada provides a cluster federation across different testbed infrastructures, as shown in the next Section, by creating a new abstraction of a federation layer. However, considering the architecture needs of the 5G-EPICENTRE, more functionalities can be further developed for an end-to-end service across 5G-EPICENTRE testbed infrastructures.

## 3.1 Architecture

The architecture presented in this Section (Figure 11) encompasses the backend layer, the federation layer where Karmada is deployed, and the scheduler housing the service placement module. By assimilating these consideration factors, such as latency and resource availability, the service placement can make intelligent choices on workload placements, ensuring that resources are allocated, while the requirements are fulfilled. Referring to deliverable D4.4, Karmada's scheduler provides advanced management of the workloads across different testbed infrastructures of 5G-EPICENTRE, and enables standalone propagation through policy Application Pro-

[1] https://karmada.io/

gramming Interface (API) for multi-cluster scheduling (placement), to fulfil the federated resource request efficiently. In this sense, it may be offering the functionalities assigned to the service placement component (see section 2.2) in 5G-EPICENTRE.

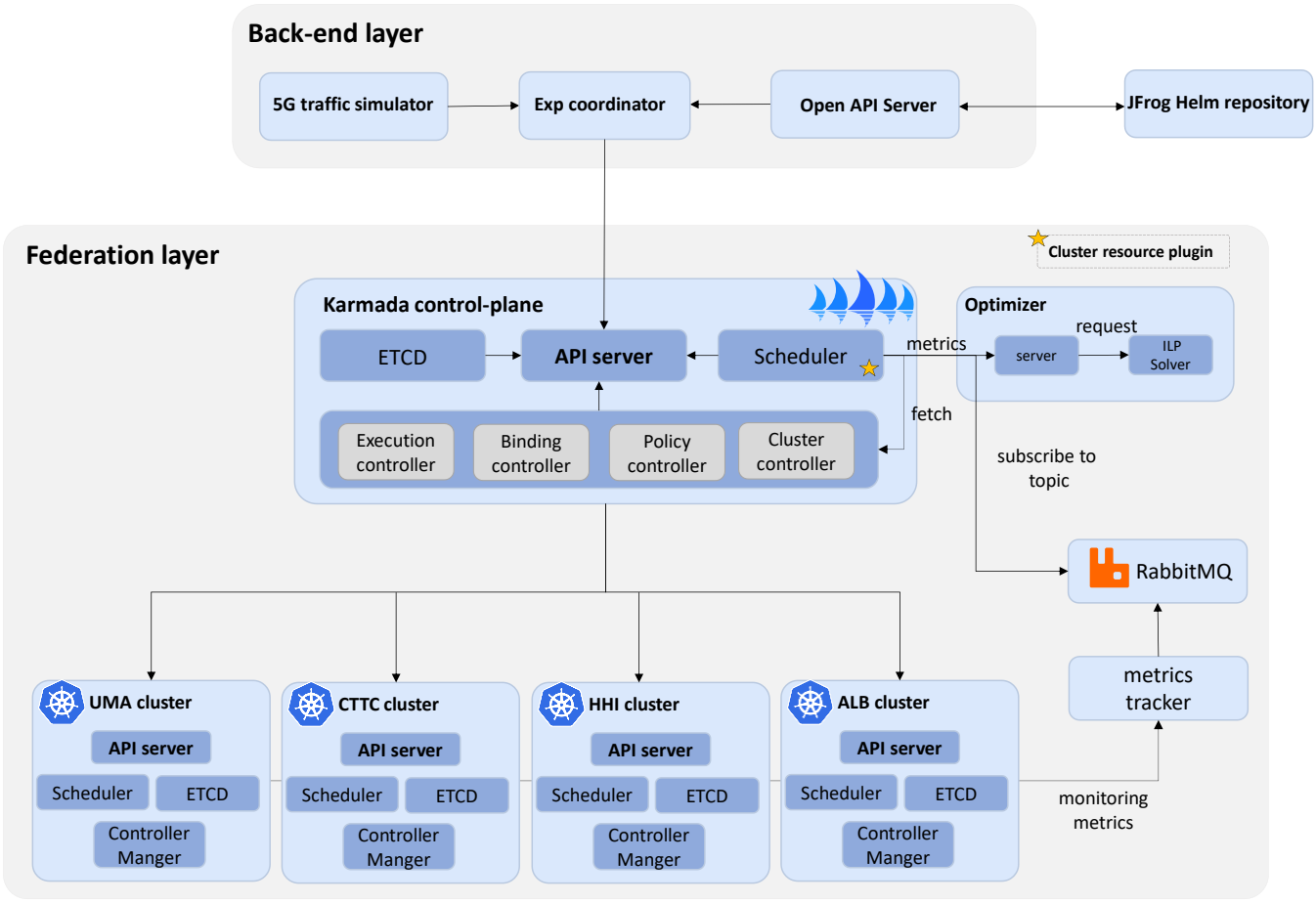



Figure 11: General architecture. This Figure depicts the main components that are connected to the scheduler, to perform the service placement.

### 3.1.1 Experiment Coordinator deployment

The **Experiment Coordinator** is the element in charge of coordinating the deployment of the experiments requested through the **Portal** in the different clusters that make up the Karmada federation. This element is located in the Backend layer, where it makes use of the different elements of that layer to carry out its work, as shown in the Figure 11. In general, the module is in charge of the coordinating the deployment of the different experiments in the selected cluster.. To perform the deployment of the experiments, the Experiment Coordinator receives from the Portal the information in the form of a JSON file (Experiment Descriptor). The Descriptor contains information about the name of the experiment to be deployed, the date and duration of the experiment and the namespace where it will be deployed. Once this information is obtained, the deployment of a **Helm repository** is obtained. In addition, it is possible to use the **5G Traffic Simulator Manager** to inject 5G traffic into the network where the experiment will be executed to simulate the different possible scenarios. The experiment execution workflow, and all of the components involved, are described in detail in deliverable D2.5.

The integration with Karmada is done by promoting the namespaces of the different clusters in the Karmada cluster. For the deployment of the different resources, the propagation policies specified for each use case must be used.

### 3.1.2 Metric measurement

In this Section, we focus on the solutions for measuring the metrics from clusters sent to a RabbitMQ message queue. As it is mentioned in D4.4, RabbitMQ acts as an intermediary or middleware, allowing components to exchange messages asynchronously and reliably. In the CTTC testbed, RabbitMQ with an MQTT plugin[2] is deployed in a testbed instance (TI), to use Message Queuing Telemetry Transport (MQTT)-type queue. The broker uses one exchange that generates queues dynamically as needed. The routing key *"application"* is used to publish experiment data to broker. The LXC VM, where RabbitMQ is installed, is based on Ubuntu 20.04 LTS. The RabbitMQ Management Dashboard is available at TCP port 15672. In the proposed architecture, RabbitMQ connects to the metric tracker modules, enabling it to publish metrics such as real-time RTT values, or aggregated available CPU resources for each cluster. Additionally, it establishes a connection with the Karmada scheduler, to enable the scheduler to fetch cloud cluster latency information for managing service placement.

#### *3.1.2.1 RTT measurements*

In this Section, the focus is on measuring RTT to a specified destination IP address, and publishing the measured RTT values to a RabbitMQ server. The solution utilizes a Python script, the *"ping"* command for RTT measurement, and the Paho MQTT library[3] for data publication. The primary objective of this RTT implementation, is to monitor network connectivity and latency to a remote machine, thereby enabling proactive troubleshooting and performance analysis.

The RTT measurement process involves executing the ping command and extracting the average RTT value from the output using a regular expression. The data publication process is also detailed, where a connection to the RabbitMQ server is established, and RTT values are published as messages to a designated topic. The measurement operates in a continuous loop, systematically measuring RTT at predefined intervals and efficiently disseminating the data to RabbitMQ for further analysis.

The *"publish.single"* function from the Paho MQTT library is used to publish the measured RTT value to the RabbitMQ server, and a topic variable is used to specify the name of the topic, where the data will be published. In this case, the topic is named *"application"*.

The *publish.single* function takes the following arguments:

- *topic*: The topic name to which the message will be published.
- *payload*: The data to be sent as the message.
- *hostname*: The hostname, or IP address of the RabbitMQ server.
- *port*: The port number used to connect to the RabbitMQ server.
- *auth*: A dictionary containing the username and password for authentication with the RabbitMQ server.

An example of the JSON message that is sent to RabbitMQ is illustrated in Figure 12. When the script executes and measures the RTT, it will send a message in this JSON format to the RabbitMQ server, allowing the scheduler to subscribe to the specified topic, and receive and process the RTT data accordingly.

### 3.1.3 Karmada and scheduler

Karmada is installed within a single K8s cluster located at CTTC. This serves as the central control plane for the federation. Using the established Virtual Private Network (VPN) connections, the K8s cluster hosted within each testbed can be seamlessly joined to the federation created by Karmada as member clusters. This means that the resources and capabilities offered by these individual testbeds become part of the larger federation.

[2] The MQTT plugin for RabbitMQ enables connecting the client to an MQTT broker using the AMQP protocol.
[3] https://pypi.org/project/paho-mqtt/

```
epicentre@epicentre-k8s-master:~/paho.mqtt.python/examples$ python3 rtt_uma.py
RTT (ping3): 25.916 ms
RTT (ping3): 26.309 ms
RTT (ping3): 26.211 ms
RTT (ping3): 26.063 ms
RTT (ping3): 26.140 ms
RTT (ping3): 26.026 ms
RTT (ping3): 26.095 ms
RTT (ping3): 32.950 ms
RTT (ping3): 26.241 ms
RTT (ping3): 26.068 ms
RTT (ping3): 26.315 ms
RTT (ping3): 26.097 ms
```

Figure 12: Example of tracking RTT which will be formatted into a JSON message and sent to the RabbitMQ server.

#### *3.1.3.1 Scheduler*

At the heart of Karmada is the **Scheduler**, a critical component responsible for determining how workloads should be distributed across multiple clusters. The importance of the scheduler in Karmada cannot be overstated. As verticals scale and deploy their applications across multiple Kubernetes clusters, possibly spanning different cloud providers and regions, efficient workload distribution becomes paramount. In addition, high availability is maintained by distributing replicas of a workload across different clusters, ensuring that a failure in one cluster does not bring down the entire application.

Basically, the scheduler monitors changes in resources via a *watch* API connected to the **Karmada API server** (refer to Figure 11). Upon detecting changes, the scheduler initializes its operations using two core components: the *cache* and the *registry*, taken from the *Cache* and runtime *Registry* interfaces, respectively. Depending on active configurations, the scheduler may invoke either the *Filter Plugin*, the *Score Plugin*, or both. Once the scheduling template is created, incorporated into the primary Karmada scheduler's plugin list, ensuring it is operational when handling new workloads. Subsequent to these steps, the *assign replica* function designates the appropriate clusters for specific resources using its replica scheduling logic. If no appropriate cluster can be found, this triggers the de-scheduling module, which is a separate component within Karmada. Concluding the process, the *Binding* method is invoked, adapting the placement, and updating the resource binding objects post-scheduling, based on the chosen placement strategy.

The Karmada scheduler, like the Kubernetes scheduler, leverages a modular framework that allows for extensibility and customization through plugins. By default, the Karmada scheduler comes with a collection of plugins that provide a set of functionalities, each operating at specific extension points (like filtering and scoring). These plugins include:

- The **Cluster Affinity** plugin in Karmada is instrumental in ensuring workloads adhere to specific affinity rules. This plugin specifically checks if a resource's selector aligns with a cluster's labels. *Labelling* is a mechanism in Karmada, that an administrator can assign a name to a cluster, to dictate which specific applications are suitable to run on a certain cluster. If a match is found, it indicates that the workload has an affinity to that particular cluster, guiding the scheduler's placement decision.
- The **API Enablement** plugin in Karmada checks the readiness and presence of the necessary API, typically a Custom Resource Definition (CRD), in the target cluster. This ensures that the cluster can support the desired resource type before any scheduling decisions are made, thereby avoiding potential incompatibilities, or failures post-scheduling.
- **Cluster Eviction** is crucial in scenarios where resources within a cluster become constrained, either due to overutilization or failures.
- **Cluster Locality** is a scoring plugin within Karmada, which prioritizes clusters that already host a given resource. By doing so, it potentially reduces the need for data transfer, maintains data locality, and can

enhance the performance of certain applications by minimizing latency and ensuring related components reside within the same cluster.

- The **Spread Constraint** plugin ensures that workloads adhere to the desired distribution properties defined in the specification of clusters (*cluster.Spec*).This plugin checks and enforces these spread properties, ensuring that workloads are distributed across clusters in a manner that aligns with predefined specifications.

#### *3.1.3.2 Cluster Resource plugin integration*

To meet the objectives detailed in Section 2, we introduced a newly crafted plugin named *cluster resource*, which integrates with the standard scheduler in the Karmada system. Its primary role is to ensure the achievement of essential KPIs, by determining the best placement for any given service. While the plugin operates independently from both the service and the infrastructure, it extracts crucial metrics, with a focus on latency, for scheduling the PPDR service. Similar to native plugins in the Karmada scheduler, the *cluster resource* plugin includes primary structure. First let us elaborate on the specific steps taken to implement a custom scheduler plugin within the Karmada environment. As a point of reference, the established karmada-scheduler implementation within "*pkg/scheduler/framework/plugins"* of the Karmada source directory was consulted [13]. Consequently, the directory structure post-development resembled:

***FilterPlugin***

- *APIEnablement*
- *ClusterAffinity*
- *ClusterEviction*
- *SpreadConstraint*
- *TaintToleration*
- *ClusterResource*
  - *cluster_resource.go*

***ScorePlugin***

- *ClusterAffinity*
- *ClusterLocality*

As shown in Figure 13, The principal file, *cluster_resource.go*, was structured with core GoLang constructs.

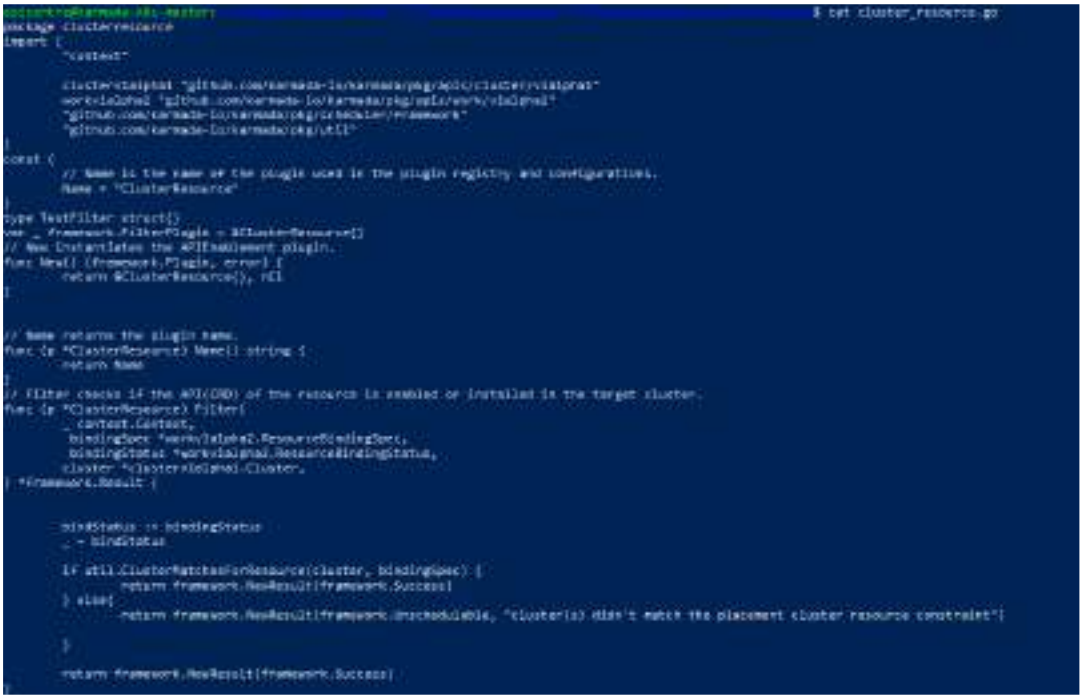


Figure 13: An example of implementation of cluster resource plugin in Karmada scheduler

Essential imports, such as context and various Karmada packages formed the basis of the code. This plugin is developed as a filtering extension, which is called during Filtering phase. As a next step, the plugin was registered within the "*cmd/scheduler/main.go"* file (Figure 14).

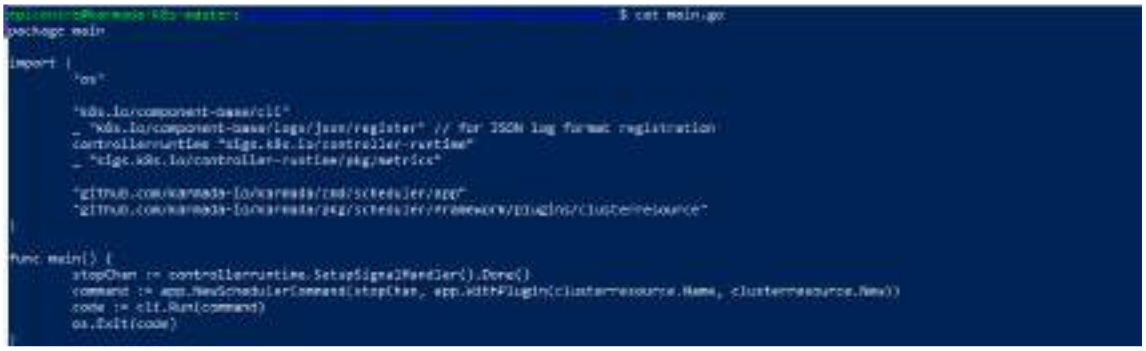


Figure 14: Plugin registration

The *NewSchedulerCommand* function, crucial for the plugin's instantiation, was edited to pass the plugin configuration. This allowed for the effortless detection and activation of the plugin during runtime. Finally, the executable code of scheduler is enclosed within a container image (Figure 15 b), and called along with other standard plugins during scheduling.

### *3.1.3.3 Cluster Resource plugin workflow*

Similar to the standard plugins, which use Filter extension points, our Cluster Resource plugin receives two APIs, *"cluster v1 alpha1"* and *"policy v1 alpha1"*, to access the essential information related to status of cluster and service requirements. On the other hand, to retrieve the CPU data of each cluster, the plugin employs both internal and external methods. It either communicates through an internal interface, or uses a client API to query

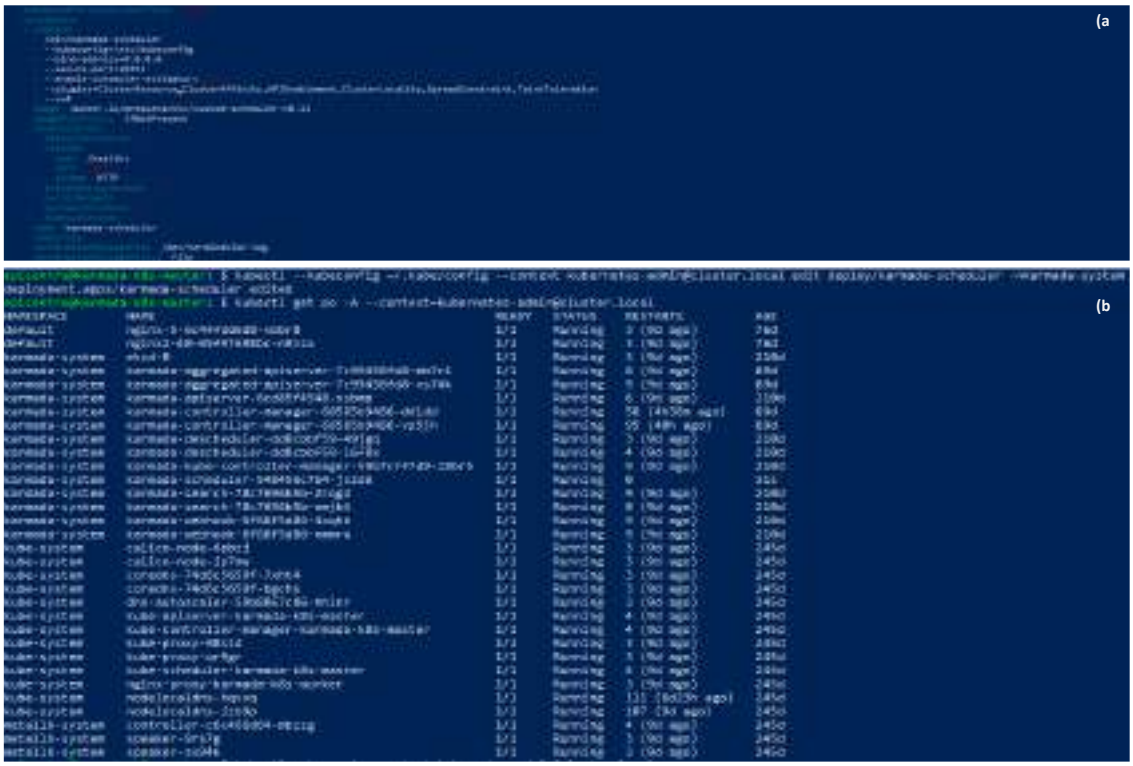


Figure 15: a) Represents the process of updating the scheduler image to include a new plugin; and b) confirms that the container has been pulled from the repository, and is running successfully in the pod.

the K8s' API server about the allocatable resources of each member cluster. Additionally, it subscribes to a RabbitMQ broker topic, to fetch cluster-to-emergency latency. We have integrated a RTT tracker module, that periodically monitors response time to the Cloud cluster, encapsulating it as *"rtt"* payload, and publishing it to RabbitMQ (Figure 12).

The Optimizer modules collaborate with the cluster resource plugin in solving service placement problems. To expedite these computations, an ILP Solver, which is presented in section 2, is in place, which might be running on a server, presumably a Flask server. Upon receiving inputs, the scheduler activates these modules to compute the optimal solution, as detailed in Section 2.2, and subsequently forwards the results back to the scheduler. The available resource for a given cluster, is the aggregated allocatable CPU of all nodes, while end-to-end (E2E) latency represents the response time from master node to the end user. The cluster resource algorithm employs a linear programming technique to tackle the placement problem, striving to minimize the maximum load of any cluster, without surpassing metric constraints. This results in efficient load balancing, when deploying multiple services at the same time, as in an emergency scenario (see Section 2 for a detailed account).

# 4 Conclusions

This deliverable is dedicated to an extensive evaluation of Task 2.3 regarding service placement strategies with a particular focus on PPDR scenarios. Our study utilized the service function (SF) concept, and employed the ILP approach to optimize the allocation of computational and network resources across various clusters. In the theoretical part, the results of our analyses highlight the superior performance of ILP in comparison to the LB method, which served as our benchmark.

The first analysis revealed that ILP consistently outperformed LB in terms of resource allocation efficiency, maintaining lower levels of unallocated CPU workloads even under high-demand conditions. This demonstrates that ILP is better suited to support critical services in emergency situations. Furthermore, our investigation into service deployments showed that ILP was capable of deploying a greater number of services compared to LB, particularly under high load conditions. This underscores ILP's effectiveness in managing service placements, ensuring that critical services are deployed without waiting in queue, which lead to minimum service creation time.

Our load distribution analysis illustrated ILP's ability to evenly distribute workload across clusters, minimizing the maximum load on any single cluster. This not only prevents bottlenecks, but also enhances the overall efficiency and performance of the system. In contrast, LB displayed a tendency to unevenly distribute load, particularly when clusters had varying capacities. This could potentially lead to suboptimal resource utilization and decreased system performance.

In conclusion, the ILP approach to service placement proves to be a robust and efficient strategy, particularly in PPDR scenarios challenging and dynamic environments. Our findings highlight the importance of intelligent service placement strategies in optimizing resource utilization, enhancing system performance, and ultimately, supporting swift and effective decision-making in critical situations.

In addition, our experimental development provides an innovative plugin integrated within the Karmada system. This plugin not only augments the Karmada system's capabilities but also offers a flexible framework that can be easily extended to integrate with other modules of project as required. By addressing service placement with a focus on critical metrics such as latency and computational resources, this novel plugin has demonstrated its suitability in meeting the stringent requirements of PPDR use cases. The successful development and integration of this plugin mark a significant step towards optimizing resource allocation and enhancing the performance of dynamic network environments.

# References


[1] Tabatabaei, F., Khalili, H., Requena, M., Kahvazadeh, S., & Mangues-Bafalluy, J. (2023, July). Dynamic service placement in 6g multi-cloud scenarios. In 2023 23rd International Conference on Transparent Optical Networks (ICTON) (pp. 1-4). https://doi.org/10.1109/ICTON59386.2023.10207547

[2] Tajiki, M. M., Salsano, S., Chiaraviglio, L., Shojafar, M., & Akbari, B. (2018). Joint energy efficient and qos-aware path allocation and vnf placement for service function chaining. IEEE Transactions on Network and Service Management, 16(1), 374-388. https://doi.org/10.1109/TNSM.2018.2873225

[3] Hawilo, H., Jammal, M., & Shami, A. (2019). Network function virtualization-aware orchestrator for service function chaining placement in the cloud. *IEEE Journal on Selected Areas in Communications, 37*(3), 643-655. https://doi.org/10.1109/JSAC.2019.2895226

[4] Wang, X., Wu, C., Le, F., Liu, A., Li, Z., & Lau, F. (2016). Online vnf scaling in datacenters. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)* (pp. 140-147). https://doi.org/10.1109/CLOUD.2016.0028

[5] Pham, C., Tran, N. H., Ren, S., Saad, W., & Hong, C. S. (2017). Traffic-aware and energy-efficient vnf placement for service chaining: Joint sampling and matching approach. *IEEE Transactions on Services Computing, 13*(1), 172-185. https://doi.org/10.1109/TSC.2017.2671867

[6] Moualla, G., Turletti, T., & Saucez, D. (2019). Online robust placement of service chains for large data center topologies. *IEEE Access, 7*, 60150-60162. https://doi.org/10.1109/ACCESS.2019.2914635

[7] Harutyunyan, D., Shahriar, N., Boutaba, R., & Riggio, R. (2020). Latency and mobility–aware service function chain placement in 5G networks. *IEEE Transactions on Mobile Computing, 21*(5), 1697-1709. https://doi.org/10.1109/TMC.2020.3028216

[8] Sonkoly, B., Szabó, R., Németh, B., Czentye, J., Haja, D., Szalay, M., et al. (2020). 5G applications from vision to reality: Multi-operator orchestration. *IEEE Journal on Selected Areas in Communications, 38*(7), 1401-1416. https://doi.org/10.1109/JSAC.2020.2999684

[9] Dräxler, S., Karl, H., Kouchaksaraei, H. R., Machwe, A., Dent-Young, C., Katsalis, K., & Samdanis, K. (2018). 5G os: Control and orchestration of services on multi-domain heterogeneous 5g infrastructures. In *2018 European Conference on Networks and Communications (EuCNC)* (pp. 1-9). https://doi.org/10.1109/EuCNC.2018.8443210

[10] Benkacem, I., Taleb, T., Bagaa, M., & Flinck, H. (2018). Optimal vnfs placement in cdn slicing over multi-cloud environment. *IEEE Journal on Selected Areas in Communications, 36*(3), 616-627. https://doi.org/10.1109/JSAC.2018.2815441

[11] Santoyo-González, A., & Cervelló-Pastor, C. (2018). Latency-aware cost optimization of the service infrastructure placement in 5g networks. *Journal of Network and Computer Applications, 114*, 29-37. https://doi.org/10.1016/j.jnca.2018.04.007

[12] IBM ILOG CPLEX Optimization Studio. Accessed: Apr. 1, 2020.[Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio/

[13] https://karmada.io/docs/developers/customize-karmada-scheduler#deploy-a-plugin