Cost-Effective Cloud-Edge Elasticity for Microservice Applications

Andrea Detti, Alessandro Favale

***Abstract*—Microservice applications, composed of independent containerized components, are well-suited for hybrid cloud–edge deployments. In such environments, placing microservices at the edge can reduce latency but incurs significantly higher resource costs compared to the cloud. This paper addresses the problem of selectively *replicating* microservices at the edge to ensure that the average user-perceived delay remains below a configurable threshold, while minimizing total deployment cost under a pay-per-use model for CPU, memory, and network traffic.**

We propose a greedy placement strategy based on a novel analytical model of delay and cost, tailored to synchronous request/response applications in cloud–edge topologies with elastic resource availability. The algorithm leverages telemetry and load balancing capabilities provided by service mesh frameworks to guide edge replication decisions.

The proposed approach is implemented in an open-source Kubernetes controller, the Geographical Microservice Autoplacer (GMA), which integrates seamlessly with Istio and Horizontal Pod Autoscalers. GMA automates telemetry collection, cost-aware decision making, and geographically distributed placement. Its effectiveness is demonstrated through simulation and real testbed deployment.

***Index Terms*—Edge Computing, Microservices Applications, Service Meshes**

I. Introduction

Microservice architectures decompose server-side applications into loosely coupled services that communicate via APIs such as HTTP or gRPC [1], [2], [3]. This modular design enhances development flexibility, scalability, and deployment velocity. Performance-wise, microservices support horizontal distribution across clusters, improving fault isolation and resource allocation.

Linux containers are the de facto standard for deploying microservices, offering lightweight, isolated environments. Kubernetes is the leading orchestration platform, automating microservice deployment, scaling, and communication. Each microservice instance runs in a *Pod*, the smallest schedulable unit in Kubernetes. Applications typically replicate Pods across nodes to support load balancing and resilience. *Horizontal Pod Autoscalers (HPAs)* monitor resource metrics, such as CPU usage, and automatically scale the number of replicas per microservice.

Complementing Kubernetes, service mesh frameworks like Istio provide security, traffic management, and monitoring [4]. They implement advanced load balancing strategies that go beyond Kubernetes' default random approach by incorporating runtime and topology metrics. In multi-cluster deployments, service meshes enable *locality-aware load balancing*, prioritizing local replicas to reduce cross data center interactions between microservices during user request processing, thereby lowering latency compared to random load balancing.

At the same time, *edge computing* is emerging as a paradigm to process data closer to users, reducing latency. 5G networks accelerate this trend by routing mobile traffic through local edge data centers via User Plane Functions (UPFs), shortening network paths [5], [6]. However, edge resources are significantly costlier than centralized cloud infrastructure, by over 30% in public cloud pricing[1]. This cost–latency trade-off calls for microservice placement strategies that balance application responsiveness with operational cost efficiency.

Motivation: While many prior works address placement under static or resource-constrained edge scenarios [7], modern edge platforms offer elastic, pay-per-use models similar to the cloud. This evolution motivates cost-aware strategies that assume scalable edge capacity. Moreover, most placement strategies assume exclusive placement for which each microservice is deployed at either the cloud or a specific edge location [8], [9], [10]. In contrast, real-world platforms increasingly adopt a cloud-edge hierarchical model, where microservices are always available in the cloud and selectively *replicated* at the edge for latency reduction [11], [12].

Contributions: This work presents a novel **analytical model** of cost and user-perceived delay for microservice applications based on request-response interactions, specifically designed for hierarchical cloud–edge topologies and extending our prior work [13]. This model is the foundation of the proposed **Subgraphs-Based Microservice Placement (SBMP)** strategy, which aims to minimize pay-per-use deployment costs while ensuring that user delay remains below a configurable threshold. SBMP employs a greedy algorithm that incrementally replicates subgraphs of microservices to edge data centers. These subgraphs are extracted from request traces, and selection is guided by delay and cost metrics derived from the analytical model.

SBMP is implemented in a new open-source Kubernetes controller, the **Geographical Microservice Autoplacer (GMA)** [14], which integrates:

- Horizontal Pod Autoscalers (HPAs) for intra-cluster replication;
- Istio service mesh for telemetry and locality-aware load balancing;
- A placement engine that activates edge replicas only when latency constraints cannot be met by HPAs alone.

Authors are with CNIT and the Department of Electronic Engineering, University of Rome "Tor Vergata", Italy, e-mail: {name.surname@uniroma2.it}

This work was partially supported by Italian PNRR funds under the RESTART program and EU FLUIDOS Project.

[1]E.g., in November 2024, AWS t3.medium edge-computing (Wavelength US-East) $0.056/hr vs. cloud region (US-West Oregon) $0.0416/hr.

GMA is fully compatible with standard Kubernetes and Istio deployments and operates on any application exposing basic telemetry, making it a practical solution for adaptive microservice replication in real-world cloud–edge systems.

II. Analytical Models

In this section, we present the analytical models that form the foundation of the SBMP placement strategy employed by the GMA controller. First, we describe the model of the considered cloud-edge system, and then we afford the analytical model of delays and costs. Table I shows the main variables used in this section.

A. System Model

1) Application Work Model: The microservice application comprises H-1 microservices, with the user modeled as a virtual Hth microservice. Upon receiving a request, microservice i executes its *internal function* and then serially invokes a stochastic subset of its downstream microservices, with the invocation order chosen at random. After all downstream responses are returned, i finalizes processing and sends a response back to its upstream caller [2] [15].

Stochastic interactions depend on the structure of the application and the user request pattern. They can be modeled by an $H \times H$ call frequency matrix F_M, where each entry $F_M(i,j)$ denotes the average number of calls made by i to j per request received by i. From this matrix, a *microservice dependency graph* G_M can be constructed: nodes represent microservices, and a directed edge from i to j exists if $F_M(i,j) > 0$. We assume G_M is a directed acyclic graph (DAG) rooted at the user [3]. For example, Fig. 1 illustrates the microservice dependency graph of an application composed of eight microservices. In the figure, the node labeled "H" represents the user, while the numbered nodes correspond to the individual microservices.

2) Infrastructure Model: The application runs on an infrastructure made of a cloud and an edge data center. Data centers provide virtually unlimited CPU and memory resources and are interconnected by a bidirectional network link with a round-trip time RTT and bitrate B. The user connects to the *ingress gateway* of the edge data center, labeled "in" in Fig. 1 and subsequent graphs [4]. Within each data center, the internal network is high-speed and low-latency, rendering intra-data center communication delays negligible.

Cloud and edge resources may differ in processing power, resulting in varied execution times. Specifically, $D_i(i_c)$ and $D_i(i_e)$ denote the internal processing time of microservice i on cloud and edge instances, respectively.

3) Deployment Model: A microservice i is deployed as a set of parallel instances (e.g., Kubernetes Pods), with i_c and i_e denoting the group of instances running in the cloud and at the edge, respectively. In the next, we also use the notation $i_{c|e}$ when discussing operations or properties that apply to both cloud and edge instances.

The binary vector S denotes the placement state: $S(i_c) = 1$ indicates that microservice i has cloud instances, while $S(i_e) = 1$ indicates the presence of edge instances[5].

Each microservice has at least one cloud instance, therefore $S(i_c) = 1 \ \forall \ i < H$. We refer to this assumption as *cloud persistence*. As the user (H) is present only to the edge, $S(H_e) = 1$ and $S(H_c) = 0$. For each $i < H$, values of $S(i_e)$ can be either 0 or 1, depending on the active placement configuration.

For example, Fig. 2 illustrates a possible deployment for the microservice application in Fig. 1. Under the cloud persistence assumption, all microservices have at least one instance in the cloud. Additionally, microservices 1, 2, and 5 also have instances running at the edge, colocated with the user.

4) Resource Reservation and Autoscaling: Each cloud or edge instance of microservice i reserves CPU and memory resources, specified by $Q_{\text{cpu}}(i_{c|e})$ and $Q_{\text{mem}}(i_{c|e})$, respectively[6].

Horizontal autoscalers in each data center adjust the instance count to ensure that the CPU usage of each instance remains below $\rho_{cpu}\, Q_{\text{cpu}}(i_{c|e})$, where ρ_{cpu}, typically between 0.5 and 0.7, is the target utilization ratio applied to the CPU quota reserved per instance.[7]

The number of cloud and edge instances (i.e., replicas) of microservice i for a placement S, denoted as $R_p(S, i_c)$ and $R_p(S, i_e)$, respectively, is computed as:

$$\begin{aligned} R_p(S,i_c) &= \max\left(\left\lceil \frac{U_{\text{cpu}}(S,i_c)}{\rho_{cpu}\, Q_{\text{cpu}}(i_c)} \right\rceil, 1\right) \\ R_p(S,i_e) &= \left\lceil \frac{U_{\text{cpu}}(S,i_e)}{\rho_{cpu}\, Q_{\text{cpu}}(i_e)} \right\rceil \end{aligned} \tag{1}$$

Here, $U_{\text{cpu}}(S,i_c)$ and $U_{\text{cpu}}(S,i_e)$ are the cumulative CPU utilization of cloud and edge instances of microservice i, respectively. The $\max$ operator ensures that each microservice always has at least one cloud instance.

5) Instance Interaction and Load Balancing: The interactions among instances under a placement state S are captured by a $2H \times 2H$ call frequency matrix $F_I(S)$, where each entry $F_I(S, i_{\{c|e\}}, j_{\{c|e\}})$ denotes the average number of times instances of $i_{\{c|e\}}$ call instances of $j_{\{c|e\}}$ per request received by $i_{\{c|e\}}$.

The matrix $F_I(S)$ can be used to build a *instance dependency graph* $G_I(S)$. A node corresponds to cloud or edge instances of a microservice, and a directed link exists between two nodes $i_{\{c|e\}}$ and $j_{\{c|e\}}$ if their corresponding

[2]For a given microservice i, a downstream microservice is one that it calls, while an upstream microservice is one that calls it.

[3]The assumption that G_M is a DAG reflects the hierarchical invocation structure typical of many microservice-based applications [16] [17]. While cycles may occur in specific cases (e.g., event-driven systems), they are generally avoided to ensure modularity and simplify fault management. Extending the model to cyclic dependencies is left for future work.

[4]This reflects typical 5G edge scenarios where user traffic is statically routed to a designated edge site, often based on long-term agreements independent of microservice placement.

[5]i_c and i_e serve as *symbolic* indices that, for computational purposes, are mapped to linear positions k in the placement vector S. In our implementation the mapping is defined as: $i_c \mapsto i$ and $i_e \mapsto H + i$. For example, a model-level condition $S(i_e) = 1$ corresponds to $S(i + H) = 1$. Similarly, a call frequency entry $F_I(i_e, j_e) = 1$ is implemented as $F_I(i + H, j + H) = 1$.

[6]For Kubernetes applications, Q_{cpu} and Q_{mem} match both `Request` and `Limit` values, as in the *Guaranteed* QoS class.

[7]Without loss of generality, we assume that all microservices adopt the same value of ρ_{cpu}.

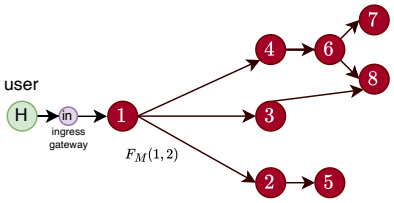

Figure 1: Microservice Dependency Graph (G_M)

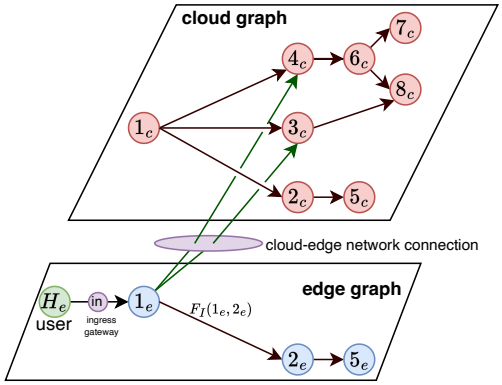

Figure 2: Instance dependency graph (G_I)

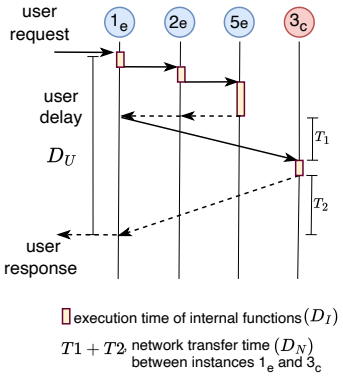

Figure 3: Trace of a user request

instances interact, i.e. $F_I(S, i_{\{c|e\}}, j_{\{c|e\}}) > 0$. Fig. 2 provides an example of an instance dependency graph G_I, related to the microservice dependency graph in Fig. 1, for which microservices 1,2, and 5 have replicas in the edge data center.

The instance-level call frequency matrix $F_I(S)$ is derived from the microservice-level matrix F_M and depends on the load balancing policy used to distribute requests among instances. We adopt a *locality-aware* load balancing, in which requests to a microservice i are preferentially routed to instances located within the same data center as the requesting instance. If no local instances are available, requests are forwarded to the cloud instances j_c as a fallback.

Under this policy, and assuming the cloud persistence constraint, Eq. 2 defines how $F_I(S)$ is computed from F_M.

$$
\begin{aligned}
&F_I(S, i_c, j_c) = F_M(i, j) \\
&F_I(S, i_c, j_e) = 0 \\
&F_I(S, i_e, j_e) = \begin{cases} F_M(i,j)\text{ , if } S(i_e) = 1 \text{ and } S(j_e) = 1 \\ 0\text{ , otherwise} \end{cases} \\
&F_I(S, i_e, j_c) = \begin{cases} F_M(i,j)\text{ , if } S(i_e) = 1 \text{ and } S(j_e) = 0 \\ 0\text{ , otherwise} \end{cases}
\end{aligned}
\tag{2}
$$

Regardless of the placement configuration, each instance of microservice i issues calls to microservice j with frequency $F_M(i, j)$. When the calling instance of i is in the cloud, it will always invoke the cloud instance of j, as cloud instances are guaranteed to exist and are preferred due to locality. This justifies the first two equations in Eq. 2. When the calling instance of i is at the edge, it will invoke the edge instance of j if it exists, and never the cloud instance, reflecting strict locality. This explains the third case in Eq. 2. If the called microservice j is not deployed at the edge, the request is routed to the cloud, as captured in the last condition of Eq. 2.

For example, Fig. 3 shows a possible sequence of instance-level interactions based on the configuration in Fig. 2, involving microservices 1, 2, 3, and 5. In this example, requests are served by edge instances whenever available.

We note that, since all microservices remain available in the cloud (cloud persistence), even when multiple edge data centers are present, locality-aware load balancing confines communication to individual edge and cloud pairs. Specifically, when an edge instance i_e (or the user) invokes a cloud instance j_c, j_c processes the request using only local cloud microservices and then replies to i_e. As a result, no edge-to-edge interactions occur. This justifies the simplified infrastructure model with a single edge and cloud data center adopted in subsubsection II-A2.

6) Cost Model: The cost model reflects typical pricing schemes used by cloud providers for containerized applications (e.g., GKE Autopilot mode) [18], [19]. Providers charge a rate C_{net} per GB of inter-datacenter traffic, and hourly rates $C_{\text{pod}}(i_c)$ and $C_{\text{pod}}(i_e)$ for the CPU (Q_{cpu}) and memory (Q_{mem}) resources reserved by each Pod (i.e., instance) of microservice i in the cloud and edge datacenter, respectively.

The total hourly cost $C(S)$ of a placement state S comprises three components: cloud resource cost $C_C(S)$, edge resource cost $C_E(S)$, and inter-datacenter network cost $C_N(S)$:

$$
\begin{aligned}
C(S) &= C_C(S) + C_E(S) + C_N(S) \\
C_C(S) &= \sum_{i_c} R_p(S, i_c) \cdot C_{\text{pod}}(i_c) \\
C_E(S) &= \sum_{i_e} R_p(S, i_e) \cdot C_{\text{pod}}(i_e) \\
C_N(S) &= 4.5 \cdot 10^{-7} \cdot C_{\text{net}} \cdot T_{NCE}(S)
\end{aligned}
\tag{3}
$$

Here, $R_p(S, i_{c|e})$ denotes the number of cloud or edge instances of microservice i, and $T_{NCE}(S)$ is the network traffic (in bit/s) sent from cloud to edge. Reverse traffic is considered negligible, as requests are typically lightweight (e.g., HTTP), and locality-aware load balancing favors local service resolution. The factor $4.5 \cdot 10^{-7}$ converts traffic from bit/s to GB/hour.

B. Analytical model of delay and cost

To optimize a placement strategy, it is essential to estimate the delay variation a new placement might introduce based on

Table I: Summary of Main Variables in Section II

Symbol	Description	Simulation value or Eq.
B	Available bitrate between cloud and edge	1 Gbit/s
$C(S)$	Total hourly cost for placement S (computing resource + network)	Eq. 3
$C_{\text{pod}}(i_e)$	Cost per hour of CPU and memory resources reserved for a edge instance (Pod) of microservice i	$0.0208\,Q_{\text{cpu}}(i_e) + 0.028\,Q_{\text{mem}}(i_e)$
$C_{\text{pod}}(i_c)$	Cost per hour of CPU and memory resources reserved for a cloud instance (Pod) of microservice i	$0.0104\,Q_{\text{cpu}}(i_c) + 0.014\,Q_{\text{mem}}(i_c)$
C_{net}	Unit cost per GB of inter-data center network traffic	0.02
$D_i^0(i_{c\|e})$	Internal processing time of microservice i at cloud or edge under unloaded conditions	not used, $D_i(i_{c\|e})$ used directly
$D_i(i_{c\|e})$	Internal processing time of microservice i at cloud or edge under current CPU load $\hat{\rho}_{cpu}(i_{c\|e})$	random uniform in 5–10 ms, equal for cloud and edge
$D_I(S)$	User delay contribution from internal processing of all microservices under placement S	Eq. 5
$D_U(S)$	User delay perceived under placement S	Eq. 4
$D_N(S)$	User delay contribution from inter-data center network data transfer under placement S	Eq. 8
$D_n(S, i_e, j_c)$	Delay from network transfer (propagation + transmission) when edge instance i_e invokes cloud instance j_c under placement S	Eq. 9
$\Delta D_I(S^{\text{old}}, S^{\text{new}})$	Variation in internal processing delay between placements S^{old} and S^{new}	Eq. 14
$\Delta D_U(S^{\text{old}}, S^{\text{new}})$	Variation in user delay between placements S^{old} and S^{new}	Eq. 14
$\Delta D_N(S^{\text{old}}, S^{\text{new}})$	Variation in network delay between placements S^{old} and S^{new}	Eq. 14
$F_I(S, i_{c\|e}, j_{c\|e})$	Instance-level call frequency from instances $i_{c\|e}$ to $j_{c\|e}$ under placement S	Eq. 2
$F_M(i, j)$	Microservice-level call frequency from microservice i to j	random uniform in 0.1–0.5 ms
i, i_c, i_e, $i_{\{c\|e\}}$	Microservice i; its cloud (i_c), edge (i_e), or generic instances ($i_{\{c\|e\}}$)	
$L(i)$	Response message size (in bits) of microservice i	random uniform in 0.5–3.5 MB
H	Total number of microservices including the user (indexed as microservice H)	120
$N_i(S, i_{c\|e})$	Average number of calls to instances $i_{c\|e}$ per user request under placement S	Eq. 7
$Pd(S, i_e, j_c)$	Application-layer round-trip delay from edge instances of i to cloud instances of j	80 ms
$Q_{\text{cpu}}(i_{c\|e})$, $Q_{\text{mem}}(i_{c\|e})$	CPU and memory resource quotas reserved per pod (i.e., instance) of microservice i	Same for cloud/edge; random uniform integer in 1–8 for $Q_{\text{cpu}}(i_{c\|e})$, $Q_{\text{mem}}(i_{c\|e}) = 2\,Q_{\text{cpu}}(i_{c\|e})$ GB
$R_p(S, i_{c\|e})$	Number of cloud or edge pods (i.e., instances) of microservice i under placement S	Eq. 1
$S(i_c)$, $S(i_e)$	Binary indicator of whether microservice i has cloud or edge instance, respectively	
S^{old}, S^{new}, S^{co}	Old, new and cloud-only placement configurations	
$T_{NCE}(S)$	Total network traffic (bit/s) sent from cloud to edge under placement S	Eq. 11
$T_d(S, i_e, j_c)$	Transmission delay of a response from cloud instances of j to edge instances of i	Eq. 10
$U_{\text{cpu}}(S, i_{c\|e})$	Cumulative CPU usage of instances of microservice i in the cloud or edge under placement S	Eq. 26, Eq. 26
$U_{\text{cpu}}(S^{\text{co}}, i_c)$	Cumulative CPU usage of cloud instances of microservice i in the cloud-only placement	$\lambda/10\times$ (random uniform in 5–40)
$U_{\text{mem}}(S^{\text{co}}, i_c)$	Cumulative memory usage of cloud instances of microservice i in the cloud-only placement	$2 \times U_{\text{cpu}}(S^{\text{co}}, i_c)$ GB
$\alpha(i_c)$	Ratio of edge-to-cloud unloaded processing time for microservice i	1
λ	User request frequency	50 req/s
$\hat{\rho}_{cpu}(i_{c\|e})$	Observed CPU utilization ratio for microservice i in cloud or edge instances	not used for simulations
ρ_{cpu}	Target CPU utilization ratio used by Horizontal Autoscalers	60%
$\rho_{NCE}(S)$	Utilization of the cloud-to-edge link under placement S	Eq. 11

metrics from the current configuration. This section outlines the process in three parts: first, evaluating user delay for a given placement; second, estimating delay variation when transitioning to a new placement; and third, assessing the associated cost variation.

1) User Delay: The average user delay $D_U(S)$ consists of two components: the delay from cloud-edge network transfers, $D_N(S)$, and the delay from processing internal functions, $D_I(S)$, as shown in Fig. 3.

$$D_U(S) = D_I(S) + D_N(S) \tag{4}$$

The internal processing delay $D_I(S)$ is computed as:

$$D_I(S) = \sum_{i_c} N_i(S, i_c) D_i(i_c) + \sum_{i_e} N_i(S, i_e) D_i(i_e) \tag{5}$$

Here, $N_i(S, i_{\{c|e\}})$ is the average number of invocations per user request for instances $i_{\{c|e\}}$, and $D_i(i_{\{c|e\}})$ is the average time taken by these instances to execute their internal function.

As in [13], we assume that: (i) requests arriving to a microservice instance follow a Poisson stream, (ii) the time needed to execute the internal function in case of an unloaded instance has a generic distribution with average $D_i^0(i_{\{c|e\}})$, (iii) there is a single serving process that serve pending requests by sharing the reserved CPU capacity equally (processor sharing) and (iv) the queue of pending requests has infinite length. Consequently, the processing of an internal function can be modeled as M/G/1 queue with processor sharing whose average delay is equal to [20]:

$$D_i(i_{\{c|e\}}) = \frac{D_i^0\left(i_{\{c|e\}}\right)}{1 - \hat{\rho}_{cpu}\left(i_{\{c|e\}}\right)} \tag{6}$$

where $\hat{\rho}_{cpu}\left(i_{\{c|e\}}\right)$ is the current CPU utilization ratio if the microservice instance.

The frequency $N_i(S, i_{\{c|e\}})$ can be determined by solving the system of equations defined by Eq. 7 for all $1 \le i, j \le H$.

$$\begin{aligned} N_i(S, i_c) &= \sum_{j_c \neq i_c, j_e} N_i(S, j_{c|e}) F_I(S, j_{c|e}, i_c) \\ N_i(S, i_e) &= \sum_{j_e \neq i_e} N_i(S, j_e) F_I(S, j_e, i_e) \\ N_i(S, H_c) &= 0, \quad N_i(S, H_e) = 1 \end{aligned} \tag{7}$$

Under the assumptions of locality-aware load balancing, persistent cloud deployment of all microservices, and negligible intra-datacenter delays, the network delay $D_N(S)$ can be expressed as:

$$D_N(S) = \sum_{i_e} N_i(S, i_e) \sum_{j_c} F_I(S, i_e, j_c) D_n(S, i_e, j_c) \quad (8)$$

where $D_n(S, i_e, j_c)$ is the average network delay between edge instance i_e and cloud instance j_c. This delay includes the application-layer round trip delay $P_d(S, i_e, j_c)$ and the transmission time $T_d(S, i_e, j_c)$ for $L(j_c)$ response bits sent from j_c to i_e:

$$D_n(S, i_e, j_c) = P_d(S, i_e, j_c) + T_d(S, i_e, j_c) \quad (9)$$

The application-layer round-trip delay P_d is zero when the instance i is not running at the edge ($S(i_e) = 0$). Otherwise, P_d depends on the protocol used for data exchange between i_e and j_c. For example, in the case of HTTP persistent connections, P_d is equal to the network round-trip time (RTT).

The transmission time T_d is also zero when the instance i is not running at the edge ($S(i_e) = 0$). Otherwise, T_d depends on the available bitrate B and the cloud-to-edge connection utilization $\rho_{NCE}(S)$, modeled as an M/G/1 queue with processor sharing [13]:

$$T_d(S, i_e, j_c) = \frac{L(j_c)/B}{1 - \rho_{NCE}(S)}, \text{ if } S(i_e) = 1; \text{ 0 otherwise} \quad (10)$$

where $\rho_{NCE}(S)$ is defined as:

$$\begin{aligned} &\rho_{NCE}(S) = \min\left(T_{NCE}(S)/B, 1\right), \\ &T_{NCE}(S) = \lambda \sum_{i_e} N_i(S, i_e) \sum_{j_c} F_I(S, i_e, j_c) L(j_c) \end{aligned} \quad (11)$$

Here, $T_{NCE}(S)$ is the total cloud-to-edge traffic rate, and λ is the average user request rate.

2) Delay Variation: Considering an old placement configuration, S^{old}, and a new one, S^{new}, we define the delay variation of the user delay (ΔD_U), internal processing delay (ΔD_I), and network transfer delay (ΔD_N) as:

$$\Delta D_U(S^{\text{old}}, S^{\text{new}}) = D_U(S^{\text{old}}) - D_U(S^{\text{new}}) \quad (12)$$

$$\Delta D_I(S^{\text{old}}, S^{\text{new}}) = D_I(S^{\text{old}}) - D_I(S^{\text{new}}) \quad (13)$$

$$\Delta D_N(S^{\text{old}}, S^{\text{new}}) = D_N(S^{\text{old}}) - D_N(S^{\text{new}}) \quad (14)$$

From Eq. 4, ΔD_U can be computed as:

$$\Delta D_U(S^{\text{old}}, S^{\text{new}}) = \Delta D_N(S^{\text{old}}, S^{\text{new}}) + \Delta D_I(S^{\text{old}}, S^{\text{new}}) \quad (15)$$

Using Eq. 5, the variation ΔD_I in processing delay can be expressed as:

$$\begin{aligned} \Delta D_I(S^{\text{old}}, S^{\text{new}}) = &\sum_{i_e} \left(N_i(S^{\text{old}}, i_e) - N_i(S^{\text{new}}, i_e)\right) D_i(i_e) \\ &+ \sum_{i_c} \left(N_i(S^{\text{old}}, i_c) - N_i(S^{\text{new}}, i_c)\right) D_i(i_c) \end{aligned} \quad (16)$$

To simplify Eq. 16, we note that the average number of calls to a microservice per user request is a property of the application structure and of the user request pattern, regardless of how the instances are distributed between cloud and edge data centers across old and new placement configurations. Therefore, we can express this relationship as:

$$N_i(S^{\text{old}}, i_c) + N_i(S^{\text{old}}, i_e) = N_i(S^{\text{new}}, i_c) + N_i(S^{\text{new}}, i_e) \quad (17)$$

Furthermore, to account for differences in processing capabilities between cloud and edge data centers, we introduce the parameter $\alpha(i_c)$, which models the heterogeneity in processing times. This parameter is defined as the ratio of the internal processing time under unloaded conditions D_i^0 for microservice i when executed at the edge to that at the cloud [8]:

$$\alpha(i_c) = \frac{D_i^0(i_e)}{D_i^0(i_c)} = \frac{D_i(i_e)}{D_i(i_c)} \quad (18)$$

The last equality comes out from Eq. 6 and the assumption that the current CPU utilization ratio $\hat{\rho}_{cpu}(i_{\{c|e\}})$ is equal to the target utilization ratio ρ_{cpu} configured in the Horizontal Pod Autoscalers:

$$\hat{\rho}_{cpu}(i_{\{c|e\}}) = \rho_{cpu}\,. \quad (19)$$

Combining Eqs. 16, 17 and 18, the variation of processing delay can be rewritten as:

$$\begin{aligned} \Delta D_I(S^{\text{old}}, S^{\text{new}}) = &\sum_{i_c} \left(\alpha(i_c) - 1\right) D_i(i_c) \\ &\left(N_i(S^{\text{new}}, i_c) - N_i(S^{\text{old}}, i_c)\right) \end{aligned} \quad (20)$$

Eq. 20 confirms that as the processing capabilities of the cloud and edge become similar ($\alpha(i_c) \approx 1$), the variation in internal processing delay ΔD_I approaches zero and the variation in user delay ΔD_U depends primarily on network:

$$\Delta D_U(S^{\text{old}}, S^{\text{new}}) \approx \Delta D_N(S^{\text{old}}, S^{\text{new}}), \text{ if } \alpha(i_c) \approx 1 \quad \forall i \quad (21)$$

The approximation in Eq. 21, referred to as the *homogeneity simplification*, streamlines placement decision-making based on delay variation by eliminating the knowledge for specific processing times D_i, which may require complex microservice profiling or tracing tools.

3) Cost Variation: To calculate the cost variation, we use Eq. 3 as follows:

$$\Delta C(S^{\text{new}}, S^{\text{old}}) = C(S^{\text{new}}) - C(S^{\text{old}}) \quad (22)$$

We assume that the current cost, $C(S^{\text{old}})$, can be directly observed through system telemetry. For estimating the cost of a potential new placement configuration, S^{new}, we note from Eq. 1 and Eq. 3 that determining $C(S^{\text{new}})$ requires estimating the cumulative CPU utilization of the cloud and edge instances of microservice i in the new placement configuration, denoted as $U_{\text{cpu}}(S^{\text{new}}, i_c)$ and $U_{\text{cpu}}(S^{\text{new}}, i_e)$, respectively.

To achieve this, we predict these values using the following equations, which are derived based on the CPU utilization of the current configuration S^{old}, assumed to be observable through system telemetry:

[8]The value $D_i^0(i_{\{c|e\}})$ can be measured using tracing frameworks such as Jaeger [21] by issuing user requests to an unloaded system; for example, each request is sent only after the completion of the previous one.

$$S^{\text{co}}(i_e) = 0, S^{\text{co}}(i_c) = 1;\ \forall\, i < H \tag{23}$$

$$S^{\text{co}}(H_e) = 1, S^{\text{co}}(H_c) = 0 \tag{24}$$

$$U_{\text{cpu}}(S^{\text{co}}, i_c) = U_{\text{cpu}}(S^{\text{old}}, i_c) + \frac{U_{\text{cpu}}(S^{\text{old}}, i_e)}{\alpha(i_c)} \tag{25}$$

$$U_{\text{cpu}}(S^{\text{new}}, i_c) = \frac{N_i(S^{\text{new}}, i_c)}{N_i(S^{\text{new}}, i_c) + N_i(S^{\text{new}}, i_e)}\, U_{\text{cpu}}(S^{\text{co}}, i_c) \tag{26}$$

$$U_{\text{cpu}}(S^{\text{new}}, i_e) = \frac{\alpha(i_c)\, N_i(S^{\text{new}}, i_e)}{N_i(S^{\text{new}}, i_c) + N_i(S^{\text{new}}, i_e)}\, U_{\text{cpu}}(S^{\text{co}}, i_c) \tag{27}$$

This approach assumes that activating edge instances redistributes processing loads between edge and cloud without changing the total processing volume, as the request rate remains constant during reconfiguration.

First, we calculate the cumulative CPU utilization $U_{\text{cpu}}(S^{\text{co}}, i_c)$ for a hypothetical cloud-only configuration S^{co} with no edge microservice (Eqs. 23, 24). As in Eq. 25, this cumulative utilization is derived by summing the CPU usage of the current configuration (S^{old}) for both cloud and edge instances, adjusting the edge component by $\alpha(i_c)$ to account for differing processing times at the edge.

Next, Eq. 26 and Eq. 27 redistribute $U_{\text{cpu}}(S^{\text{co}}, i_c)$ across new cloud and edge instances, scaled by their proportional involvement per user request, i.e., $N_i(S^{\text{new}}, i_{\{c|e\}})/\left(N_i(S^{\text{new}}, i_c) + N_i(S^{\text{new}}, i_e)\right)$. The factor $\alpha(i_c)$ accounts for edge-cloud performance heterogeneity.

III. Greedy Placement Algorithm

A. Problem Statement

The analytical models for delay and cost variation allow us to formulate the following optimization problem: determine a new microservice placement S^{new} that satisfies a target delay variation δ in the most cost-effective way, under a *monotonic* adjustment constraint. This constraint implies that when a delay reduction is required, the set of edge microservices can only be extended; conversely, when a delay increase is acceptable, the set can only be reduced. This ensures smooth configuration transitions by avoiding simultaneous offloading and unoffloading operations at the edge.

$$\begin{aligned}
&\arg\min_{S^{\text{new}}} \quad \Delta C(S^{\text{new}}, S^{\text{old}}) \\
&\text{subject to:} \\
&\text{c1:} \quad \Delta D_U(S^{\text{old}}, S^{\text{new}}) \geq \delta \\
&\text{c2:} \quad S^{\text{new}}(H_e) = 1,\ S^{\text{new}}(H_c) = 0,\ S^{\text{new}}(i_c) = 1\ \forall\, i < H \\
&\text{c3:} \quad S^{\text{new}}(i_e) = 1 \text{ if } \delta > 0 \text{ and } S^{\text{old}}(i_e) = 1 \\
&\text{c4:} \quad S^{\text{new}}(i_e) = 0 \text{ if } \delta < 0 \text{ and } S^{\text{old}}(i_e) = 0 \\
&\text{c5:} \quad S^{\text{new}}(i_e) = 0 \text{ if } i \in \Phi
\end{aligned} \tag{28}$$

The objective is to minimize the cost variation $\Delta C(S^{\text{new}}, S^{\text{old}})$ while ensuring that the delay improvement satisfies $\Delta D_U(S^{\text{old}}, S^{\text{new}}) \geq \delta$ (constraint c1). Constraint c2 places the user at the edge and enforces cloud persistence. Constraints c3 and c4 enforce monotonicity: offloading is applied when $\delta > 0$, and unoffloading when $\delta < 0$. Constraint c5 excludes from edge placement any microservices belonging to the cloud-restricted set Φ.

The problem is a Binary Integer Nonlinear Programming (BINLP) instance, where $S^{\text{new}}(i_e) \in \{0, 1\}$, and is NP-complete [22]. We therefore propose a suboptimal greedy algorithm for its resolution.

B. Subgraph-Based Microservice Placement

A *necessary condition* for efficient edge placement is that if a microservice i is deployed at the edge, at least one of its upstream microservices must also be at the edge. Otherwise, locality-aware load balancing will continue to route traffic to the cloud, rendering the edge instance of i ineffective [13].

This condition implies that edge-placed microservices must form a connected subgraph in the dependency graph G_M, rooted at the user. We refer to this as *edge graph connectivity*.

Based on this principle, we define two greedy algorithms: Subgraph-Based Microservice Placement - Offload (SBMPo) and Subgraph-Based Microservice Placement - Unoffload (SBMPu). SBMPo is applied when $\delta > 0$ (delay reduction), and SBMPu when $\delta = -\xi < 0$ (delay increase).

1) SBMPo: Algorithm 1 implements SBMPo. Given S^{old}, a delay reduction target δ, the microservice graph G_M, and the list of cloud restricted microservices Φ, it iteratively builds S^{new} by expanding the edge graph using cost-effective subgraphs $\pi_e \in \Pi_e$ generated as described in Section III-B3.

Each candidate subgraph is evaluated using a cost-benefit ratio w, defined as the cost increase over the delay reduction, bounded by δ. The subgraph with the lowest w is selected at each iteration and its microservices added at the edge until $\Delta D_U(S^{\text{old}}, S^{\text{new}}) \geq \delta$ or no improvement is possible.

Algorithm 1 SBMP - Offload

function SBMPo($\delta, S^{\text{old}}, G_M, \Phi$)
  $S^{\text{new}} \leftarrow S^{\text{old}}$; $S^{\text{opt}} \leftarrow S^{\text{new}}$, $w_{\min} \leftarrow \infty$
  while $\Delta D_U(S^{\text{old}}, S^{\text{new}}) < \delta$ **do**
    $\Pi_e \leftarrow$ create expanding subgraphs of S^{new}
    for each $\pi_e \in \Pi_e$ **do**
      $S^{\text{t}} \leftarrow S^{\text{new}}$; $S^{\text{t}}(i_e) \leftarrow 1\ \forall i \in \pi_e$
      if $\Delta D_U(S^{\text{old}}, S^{\text{t}}) \leq 0$ **then continue**
      $w \leftarrow \Delta C(S^{\text{t}}, S^{\text{old}}) / \min(\Delta D_U(S^{\text{old}}, S^{\text{t}}), \delta)$
      if $w < w_{\min}$ **then** $S^{\text{opt}} \leftarrow S^{\text{t}}$
    end for
    if $S^{\text{new}} == S^{\text{opt}}$ **then break**
    $S^{\text{new}} \leftarrow S^{\text{opt}}$
  end while
  return S^{new}
end function

Fig. 4 shows a step-by-step example where SBMPo expands the edge graph over two iterations. The initial state has no edge microservices. In the first step, microservices $\{1, 2, 5\}$ are offloaded. In the second step, $\{3, 4\}$ are added, satisfying the delay constraint.

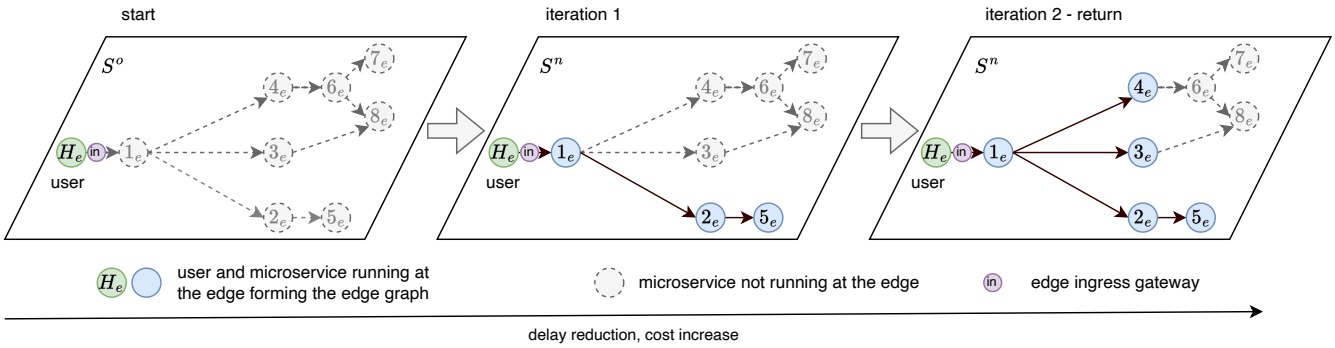

Figure 4: Evolution of edge graph during a progressive deployment of edge microservices.

2) SBMPu: SBMPu deactivates edge microservices to reduce cost while ensuring that the resulting delay increase does not exceed a threshold ξ. It reuses SBMPo by starting from a cloud-only placement S^{co} and re-adding at the edge only those microservices already present at the edge in the current configuration S^{old}, until the delay reaches the target value $D_U(S^{\text{old}}) + \xi$, where $D_U(S^{\text{old}})$ is the current user delay. The cloud-restricted set Φ is set to include all microservices not present at the edge in S^{old}, thus enforcing the monotonicity constraint: no new microservice can be added to the edge during unoffloading.

Algorithm 2 SBMPu

1: **function** SBMPU(ξ, S^{old}, G_M)
2: $\Phi \leftarrow \{i : S^{\text{old}}(i_e) = 0\}$; $S^{\text{co}} \leftarrow$ cloud-only state
3: $\delta^{co} \leftarrow D_U(S^{\text{co}}) - (D_U(S^{\text{old}}) + \xi)$
4: $S^{\text{new}} \leftarrow$ SBMPo($\delta^{co}, S^{\text{co}}, G_M, \Phi$)
5: **return** S^{new}
6: **end function**

3) Trace-Driven Expanding Subgraphs: A key component of SBMPo is the construction of expanding subgraphs Π_e (line 5, Algorithm 1). We adopt a trace-driven approach to efficiently identify promising candidate subgraphs.

Each request trace θ is mapped to a connected subgraph of G_M rooted at the user node. To control expansion granularity and enable smooth edge growth, the subgraph is pruned to limit its size.

Pruned subgraphs θ_p are derived by excluding microservices j according to the following criteria: (i) their distance $P(G_M, S, j)$ from the current edge-deployed microservices exceeds a configurable expansion depth γ; (ii) they are restricted to cloud-only deployment Φ; or (iii) they are disconnected from the user node.

A pruned subgraph θ_p is added to the candidate set Π_e if it introduces at least one new edge-deployable microservice and is not already present in the set.

Request traces can be extracted from distributed tracing tools such as Jaeger [21], or synthetically generated using the microservice-level call frequency matrix F_M.

Algorithm 3 Trace-based Expanding Strategy

1: **function** TDSP(S, G_M, γ, Θ, Φ)
2: $S_e \leftarrow$ List of edge microservices in S
3: $P(G_M, S, j) \leftarrow \begin{cases} \min\limits_{i \in S_e} \text{dist}_{G_M}(j, i), & j \notin S_e \\ 0, & \text{otherwise} \end{cases}$
4: $\Theta \leftarrow \Theta \cup \{\theta^f\}$ *# fallback trace with all microservices*
5: $\Pi_e \leftarrow \emptyset$
6: **for each** trace $\theta \in \Theta$ **do**
7: Create trace θ_p by pruning nodes $j \in \theta$ with
8: - $P(G_M, S, j) > \gamma$ **or** $j \in \Phi$
9: - $j \in \Phi$
10: - j not connected to user
11: **if** θ_p contains not-edge micros and $\theta_p \notin \Pi_e$ **then**
12: $\Pi_e \leftarrow \Pi_e \cup \{\theta_p\}$
13: **end if**
14: **end for**
15: **return** Π_e
16: **end function**

IV. Simulation Analysis

We evaluate the SBMP strategy through Python simulations, using Eq. 4 to compute user delay. For comparison, we consider four strategies: MFU, TRL, TPS, and CO-PAMP. Similarly, with SBMP , these strategies rely on runtime parameters available from Kubernetes clusters with Istio service mesh. This ensures fair comparisons among strategies that can be implemented on the same information base.

In the next subsections, we describe the algorithms used by these strategies to select the best microservice to offload/unofoload in case a delay reduction/increase is needed.

A. Baseline Strategies

Most Frequently Used (MFU): MFU reduces delay by offloading the most frequently used microservices to the edge, while it increases delay by removing the least frequently used ones. The usage frequency is measured as the number of time a microservice is called per user request.

Traffic-Locality (TRL): In [23], the authors propose a network-aware scheduler plugin that is applied at the deployment time of a microservice i, and potentially re-applied

upon changes in network conditions. The scheduling policy prioritizes nodes that host microservices with which i exchanges the highest volume of traffic. Inspired by this scoring mechanism, we define a *traffic-locality* strategy (TRL) that aims to preserve traffic locality. When a delay reduction is required, the microservice i with the highest traffic exchange with microservices already deployed at the edge—including the user—is offloaded to the edge. In the case of a tie, the candidate is selected at random. For unoffloading, the microservice i with the lowest traffic exchange with other edge-deployed microservices is removed; ties are broken randomly.

Topology-Sorting (TPS): In [9], the authors propose a network-aware scheduling framework that is applied at the deployment time of a microservice application. The scheduling policy prioritizes microservice deployment according to a topological order derived from the microservice dependency graph, computed using Kahn's algorithm for topological sorting. Inspired by this approach, we define a *topology-sorting* strategy in which, when delay reduction is needed, the microservice i with the highest topological rank among those not yet deployed at the edge is selected for offloading. In the presence of multiple candidates with equal rank, one is chosen randomly. For unoffloading, the edge microservice i with the lowest topological rank is removed, again with random selection in case of ties.

Cost-Oriented PAMP (CO-PAMP): The PAMP strategy proposed in [13] minimizes user delay under limited edge resources by progressively offloading microservices along selected *dependency paths*. A dependency path is defined as a sequence of microservices in the dependency graph G_M that connects a given microservice i to the user. Building on this concept, the proposed CO-PAMP strategy uses dependency paths to construct the set of candidate subgraphs Π_e employed in Algorithm 1.

B. Simulation Scenario

Simulations use microservice dependency graphs generated via the Barabási–Albert (BA) scale-free model [13] with power parameter equal to 0.9 and zero appeal equal to 3.125. The number of parent nodes added at each iteration is modeled as a Pareto random variable with shape $s = 10$ and a minimum value of 1, bounded by the number of existing nodes. By changing s, we can vary graph connectivity. Lower values of s result in more parent nodes, leading to a dependency graph with increased relationships among microservices.

Parameters such as call frequencies (F_M), CPU usage (U_{cpu}), memory usage (U_{mem}), response size (L), and internal delays (D_i) are sampled from uniform distributions, as summarized in Table I. Internal delays are scaled to yield a baseline user delay of 50 ms under full-edge deployment. The initial placement S^{old} is cloud-only; strategies are applied to meet a target user delay not exceeding 100 ms. Traces for SBMP are synthetically generated using F_M, which can be extracted transparently by service meshes like Istio without modifying application code or requiring tracing support. The SBMPo strategy uses an expansion depth of $\gamma = 2$. Results are averaged over 50 trials. All 95% confidence intervals are narrower than 6% of the corresponding mean.

C. Simulation Results

1) Cold-start: We first consider a *cold-start* scenario in which all strategies begin from a cloud-only placement with no edge microservices. Each greedy algorithm executes until the user delay falls below 100 ms.

Figures 5, 6, and 7 show how strategies perform as the application size grows. For dependency graphs with shape parameter $s = 10$, Fig. 5 shows the cost variation, i.e., the difference between the cost of the final placement found by each algorithm and that of the initial cloud-only deployment. SBMP consistently yields the lowest cost. As the number of microservices increases, satisfying the delay constraint becomes costlier, requiring more edge deployments (Fig. 6).

SBMP and CO-PAMP achieve the delay target at minimal cost by explicitly considering both delay and cost in each greedy iteration. CO-PAMP compares microservice *chains* for expanding the edge graph, while SBMP generalizes to multi-branch subgraphs, enabling broader and more flexible optimization.

By contrast, MFU and TRL ignore cost but outperform TPS by prioritizing frequently used microservices. Although their metrics differ, both reduce delay more effectively with fewer edge replicas, indirectly lowering cost.

TPS performs worst, as it ranks microservices by topological depth without considering their involvement per user request or cost. It may offload low-impact services, requiring more deployments to meet the delay target and increasing cost.

For more connected microservice dependency graphs ($s = 2$), Fig. 7 shows higher costs than $s = 10$, due to the need to offload more services to preserve latency, reducing the performance gap between CO-PAMP and SBMP .

Fig. 8 explores how cost varies with the target user delay. Tighter delay constraints require more edge replicas, leading to higher costs. SBMP consistently achieves the lowest cost. As the delay target approaches extreme values, all strategies converge either to a cloud-only placement (no cost variation) or a full-edge placement (maximum cost variation).

Figs. 9 and 10 show that worse network conditions (higher RTT, lower bandwidth) drive up costs by requiring more edge deployments. SBMP maintains the lowest cost in all cases.

2) Warm-start: We next consider a *warm-start* scenario that mimics online operation, where strategies maintain user delay within the 120–80 ms range. An offload is triggered when delay exceeds 120 ms (offload threshold), and unoffloading occurs when it falls below 80 ms (unoffload threshold).

During offloading, microservices are incrementally added at the edge until the delay drops below 100 ms. Conversely, during unoffloading, edge replicas are progressively removed until the delay can no longer be kept under 100 ms. No action is taken while delay remains within the 120–80 ms range.

Figures 11, 12, and 13 show results as the request rate increases from 40 to 500 req/s and back. Offloading occurs at high load; unoffloading occurs as load decreases.

Fig. 11 shows cost evolution relative to the baseline, confirming that SBMP minimizes cost across both offloading and unoffloading phases. Fig. 12 confirms that all strategies maintain delay within the requested range. Differences arise from the order in which microservices are (un)offloaded. Even with

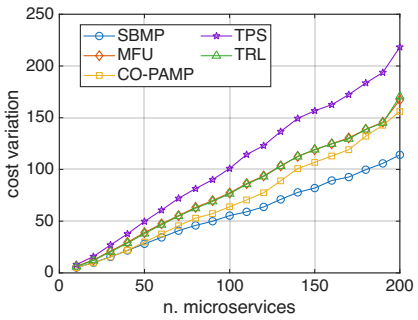

Figure 5: Cost variation relative to cloud-only placement vs number of application microservices for $s = 10$

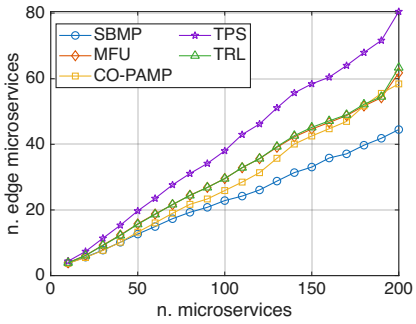

Figure 6: Number of microservices running at the edge vs number of application microservices for $s = 10$

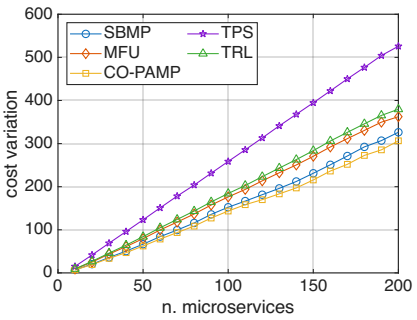

Figure 7: Cost variation relative to cloud-only placement vs number of application microservices for $s = 2$

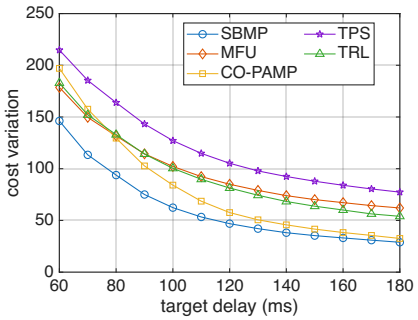

Figure 8: Cost variation relative to cloud-only placement versus target user delay

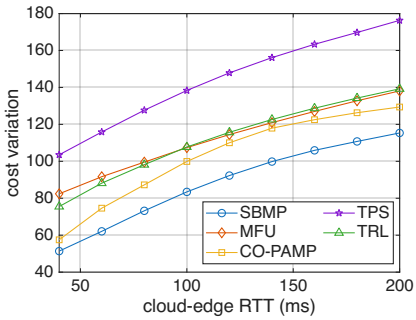

Figure 9: Cost variation relative to cloud-only placement versus cloud-edge RTT

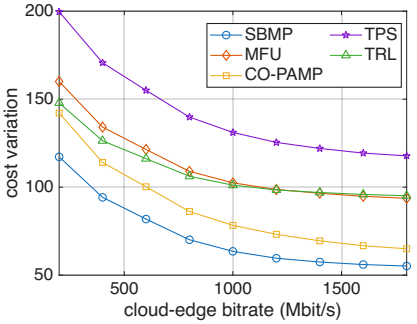

Figure 10: Cost variation relative to cloud-only placement versus cloud-edge bitrate

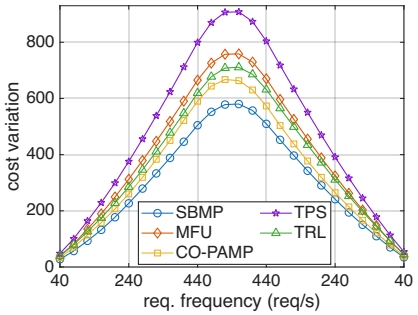

Figure 11: Cost variation relative to cloud-only placement versus request rate - warm-start

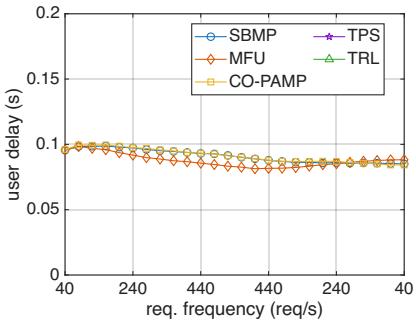

Figure 12: User delay after possible offload/unoffload action versus request rate - warm-start

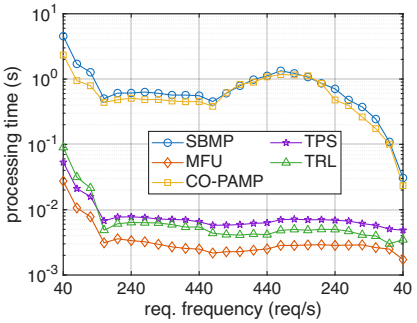

Figure 13: Processing time versus request rate - warm-start

the same request rate, strategies may result in different delays due to hysteresis between offload and unoffload thresholds.

Fig. 13 highlights the higher computational complexity of SBMP and CO-PAMP. For applications with 120 microservices, their runtime is on the order of a few seconds, compared to milliseconds for simpler heuristics. This overhead stems from evaluating multiple candidate subgraphs per step, which improves cost efficiency at the expense of additional computation. Further overhead arises during the unoffloading phase, where new configurations are recomputed from a cloud-only state (Algorithm 2), requiring more greedy iterations.

Nevertheless, the processing time remains acceptable: our Python implementation runs on a single-core Intel Xeon at 3 GHz with approximately 300 MB of RAM, and can be significantly reduced using compiled languages and parallelism. Moreover, cloud/Kubernetes control loops operate on minute-scale intervals. For example, the widely used Prometheus monitoring system uses a 60 s metric scrape interval, and the default Horizontal Pod Autoscaler (HPA) of Kubernetes scale-down stabilization period is 300 s. Therefore, a few seconds of placement computation is acceptable in practice.

Finally, note that the tested application with 120 microservices is already large. Most organizations run fewer than 100 [24]. For very large applications (e.g., 500+ microservices), further optimization would be needed, as current Python implementation may reach a few minutes.

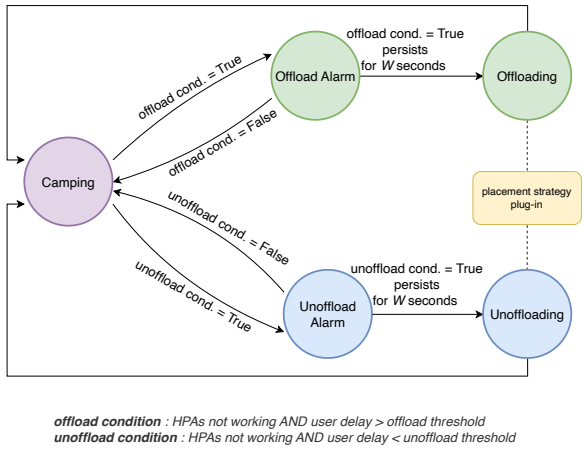

Figure 14: State machine of the GMA controller.

V. Geographical Microservice Autoplacer

The Geographical Microservice Autoplacer (GMA) is an open-source Kubernetes controller that maintains user-perceived delay within a configurable range, bounded by offload (upper) and unoffload (lower) thresholds.

GMA operates alongside Horizontal Pod Autoscalers (HPAs). When HPAs alone cannot maintain delay within bounds, GMA invokes a plug-in module that executes a placement strategy. Although multiple strategies are supported, this section evaluates SBMP only. For brevity, we provide a high-level overview; full details are available in [14].

A GMA configuration file defines thresholds, cost parameters, cluster credentials, RTT, bandwidth, Kubernetes manifests, and Istio services. With this minimal input, GMA integrates with Kubernetes to manage Deployments, Services, and HPAs, and collects metrics from Istio/Prometheus for the placement strategy. These include the current placement, user delay, CPU and memory resource usage, and the call frequency matrix F_M. Istio also provides locality-aware load balancing.

Fig. 14 shows the GMA state machine. In the *Camping* state, the controller monitors metrics and HPA activity. If delay crosses a threshold and the HPA is inactive, a placement update is required. GMA then enters an alarm state. If the condition persists for W seconds, GMA triggers an offload or unoffload action as recommended by the placement strategy, then returns to *Camping*.

During offloading, all required edge replicas (from Eq. 1) are proactively instantiated. Similarly, during unoffloading, cloud replicas are reactivated to replace dismissed edge instances. This proactive replication mitigates delay spikes caused by reactive HPA scale-out.

In multi-edge deployments, a separate GMA instance should be deployed for each edge–cloud pair. For each edge data center k, the local request rate λ_k is measured. The value $U_{\text{cpu}}(S^{\text{old}}, i_c)$ used in Eq. 25 is obtained by scaling the cumulative CPU usage of cloud instances of microservice i by the ratio $\lambda_k / \sum_k \lambda_k$, since cloud instances are shared across all edge sites.

VI. Experimental Analysis

We evaluate the performance of GMA and SBMP on a testbed with two Kubernetes clusters running Istio (for telemetry and locality-aware load balancing) and Liqo (for multi-cluster microservice management) [25]. Each cluster includes one master and four worker nodes, all with 8 vCPUs and 16 GB RAM. A router node emulates edge–cloud separation using Linux Traffic Control, limiting bandwidth to 100 Mbit/s and RTT to 80 ms [26][27].

The benchmark application, generated using μBench [15], consists of 10 microservices with the dependency graph shown in Fig. 15. Each interacting pair has a uniform call frequency of $F_M(i, j) = 0.3$. Response sizes $L(i)$ follow a negative exponential distribution with means of 1000 kB for microservice 0 and 100 kB for the others. Each microservice instance has the same value of CPU Request and Limit equal to $Q_{\text{cpu}} = 1$ CPU. The HPA threshold is set to $\rho_{cpu} = 0.6$.

Requests are generated by at a constant rate $\lambda = 10$ req/s using, targeting microservice 0 via the edge ingress gateway.

The baseline user delay, measured directly in the cloud under low load, is 110 ms and represents the minimum achievable delay. Since user traffic enters through the edge gateway, the actual delay increases due to edge–cloud communication and potential load. GMA controls user delay by offloading microservices to the edge based on SBMP decisions. We configure the offload and unoffload thresholds to 250 ms and 150 ms, respectively—well above the baseline but within typical limits for interactive applications [28]. The 100 ms gap between thresholds introduces hysteresis, preventing oscillatory placement adjustments.

Metrics are aggregated using a 2-minute moving average. GMA polls every 30 seconds, with an Alarm state stabilization period of $W = 2$ minutes.

A. Cold-Start Experiment

Fig. 16 shows results from a cold-start scenario. Initially, the application runs only in the cloud with one replica per microservice. At $t = 0$, user traffic begins at 10 req/s. Since there are no edge replicas, requests are routed to the cloud, overloading microservices 0, 1, 2, 3, and 9. Cloud HPAs respond by replicating overloaded microservices, but delays temporarily spike to several seconds due to CPU contention during the replication process. Once replication terminates, delay drops to 350 ms, still above the GMA offload threshold.

GMA, after detecting HPA inactivity, transitions to the Offload Alarm state and then to Offloading. It offloads microservices 0, 1, 2, and 9, reducing the delay to approximately 200 ms, within the acceptable range. The controller then returns to the Camping state. Note that HPAs alone could not meet the delay requirement, as they operate solely on local resource metrics and cannot account for network latency, which GMA explicitly addresses through edge replication.

B. Warm-Start Experiment

Fig. 17 reports results in a dynamic network setting, where RTT increases linearly from 0 to 400 ms in the period 0–2500 s

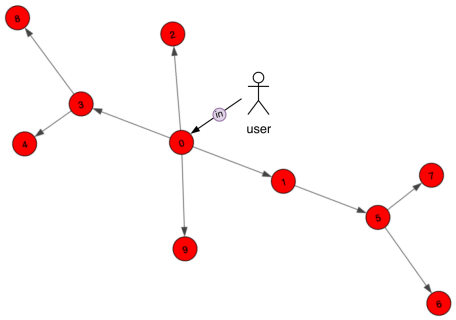

Figure 15: Microservice dependency graph of the testbed application.

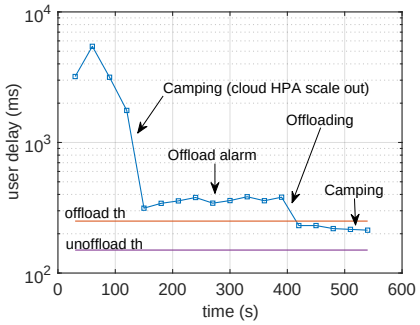

Figure 16: Results of the cold-start experiment.

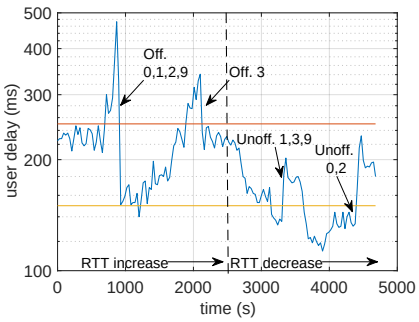

Figure 17: Results in a dynamic network environment.

and then decreases back to 0 ms. The request rate remains fixed at $\lambda = 10$ req/s. Initially, cloud-only placement suffices. At $t = 700$ s, increased RTT causes delay to breach the offload threshold. GMA triggers Offload Alarm, then offloads microservices 0, 1, 2, and 9 to the edge, restoring acceptable delay. At $t = 1900$ s, further RTT increase causes a second breach; microservice 3 is then offloaded. As RTT decreases after $t = 4000$ s, delay falls below the unoffload threshold. GMA responds by progressively unoffloading microservices 1, 3, and 9, followed by 0 and 2. This restores the original cloud-only configuration, avoiding unnecessary edge costs.

VII. Related Works

The growing adoption of edge computing has sparked significant interest in deploying microservices closer to end-users to reduce latency and improve responsiveness [37]. However, most existing approaches focus on using distributed cloud or edge resources to perform *exclusive* placement of microservices, i.e., selecting whether each microservice runs at the cloud or at any edge, but not both. To the best of our knowledge, this work is the first to address the microservice placement problem by optimizing the *replication* of microservices to edge data centers, while maintaining a full application deployment in the central cloud. Our strategy jointly minimizes operational cost and ensures user-perceived delay remains below a configurable threshold.

Furthermore, our approach uniquely integrates several practical features into the placement framework: (i) Kubernetes Horizontal Pod Autoscalers (HPAs) for dynamic resource provisioning; (ii) a pay-per-use cost model for CPU, memory, and network; (iii) service mesh functionality for locality-aware load balancing; and (iv) telemetry-driven decision making based on data exported via Istio and Prometheus. These elements allow our solution to operate within realistic cloud-native environments and to be readily deployable using standard tools as in our open-source implementation [14].

In [10], the authors introduce the Kubernetes-edge-scheduler to manage latency-critical services by placing Pods on nodes close to the request origin and migrating other Pods when resources are unavailable. In contrast, we focus on minimizing the delay of an entire application with interacting microservices, rather than individual Pods or services.

Pogonip [30] addresses the placement of asynchronous microservices interacting through a dedicated message queue. Their edge-aware scheduler prioritizes placing microservices near the message queue to limit delays. They deplete edge resources before using the cloud as a fallback. In contrast, we consider applications with distributed, peer-to-peer interactions rather than those mediated by a central message queue service. Our approach ensures overall user-perceived delay while incorporating a pay-per-use cost model based on CPU usage, rather than a node-based cost model.

The work in [31] proposes an online placement algorithm for dependency graphs of microservices structured as trees, aiming to achieve infrastructure load balancing. Their objective is to minimize the maximum weighted cost on physical nodes and links. Conversely, our strategy minimizes average user delay and edge resource utilization, adapting to general microservice interaction graphs. Additionally, we consider cloud and edge deployments and the call probabilities between microservices, which are omitted in their study.

In [32], the authors propose a learning-based deployment algorithm for microservices with linear dependency graphs, modeled as a Markov decision process. The algorithm places microservices on edge servers or migrates them based on user mobility. Differently, we focus on minimizing user delay through runtime parameters without restricting placement strategies to two options or accounting for user mobility.

The authors of [33] address task-level scheduling, where tasks are imported from the cloud for execution at the edge if unavailable locally. Their work focuses on single-request tasks rather than multi-request microservice instances, overlooking processor sharing and network utilization. In contrast, our work addresses microservice-specific aspects, such as shared usage of network resources, the presence of HPAs, and distributed interactions.

The iterative learning algorithm proposed in [34] minimizes latency for the worst-performing microservice by assigning each microservice to a single fog node. In contrast, our approach allows microservices to be concurrently deployed on both cloud and edge nodes, and focuses on minimizing the overall user-perceived delay across the entire application.

In [35], the authors propose the Reward Sharing Deep Q-Learning (RSDQL) algorithm to minimize communication overhead and balance the load among edge nodes using an interaction graph. While we also leverage an interaction graph, our work differs by minimizing user cost with bounded delay,

Table II: Comparison of Placement Strategies

Strategy / Work	Primary Objective	Cost Minimization	Edge Replication	Application Awareness
SBMP	Cost-efficient placement with delay bound	yes	yes	yes, uses dependency graph and call frequency
PAMP [13]	Delay reduction under edge constraints	no	no	yes, based on dependency paths and call frequency
Karmada [29]	Policy-based multi-cluster federation	no	no	no, pod/task-level policies
ge-kube [8]	Geo-aware placement with local elastic scaling	yes	no	partial, no dependency graph for network placement, but places groups of pods together
Diktyo [9]	Network-aware pod scheduling with SLOs	no	no	yes, considers communication topology
L. Toka [10]	Scheduling latency-critical workloads at the edge	no	no	no, per-pod latency only
Pogonip [30]	Scheduling near edge message queues	no	no	partially, for queue-based systems
Wang et al. [31]	Load-balanced placement in edge trees	no	no	yes, tree-structured dependencies
Ray et al. [32]	MDP-based placement with mobility	no	no	no, sequential placement decisions
Liu et al. [33]	Task placement with function configuration	no	no	yes, function-level dependency modeling
Nath et al. [34]	Heuristic fog container placement	no	no	no, task-level placement
Lv et al. [35]	Load-balanced fog deployment via DQL	no	no	yes, interaction-aware model
Wang et al. [36]	Vehicular delay/resource minimization	no	no	yes, per-chain constraints

with a specific focus on cloud-edge environments and a pay-per-use cost model.

The ge-kube framework [8] employs a two-step control loop: a reinforcement learning component adapts the number of local replicas based on measured response time, and a network-aware placement heuristic selects a single data center for each microservice instance by evaluating network delay between candidate sites and already-placed replicas [8]. Importantly, inter-service dependencies or call graphs are not explicitly modeled in their placement decision; instead, the heuristic selects the candidate site (from a latency-sorted list) that meets a fixed delay threshold for all site actually involved.

In [36], the authors propose a placement strategy for a hierarchical resource management framework tailored for Internet of Vehicles (IoV) environments. Their approach focuses on minimizing resource consumption and latency by leveraging mobile edge computing and cloud computing datacenters. Unlike our work, they do not account for costs, assume limited edge computational resources, and apply the delay constraint to each individual microservice in a vehicular request chain rather than to the overall delay experienced by the user.

Diktyo [9] introduces a set of network-aware scheduling plugins for Kubernetes, designed to minimize communication delay between interacting services based on a network topology model. While their system accounts for communication dependencies, their objective is not cost minimization, and they do not support microservice replication across cloud and edge as we do.

The work in [23] proposes a network-aware container scheduler for Kubernetes that takes into account both current network conditions and inter-service traffic patterns by scoring nodes based on latency and communication volume, and further uses a descheduler to reassign pods dynamically. While this design acknowledges microservice dependencies, it operates exclusively at the pod-placement level and does not support microservice replication across cloud and edge replicas as our approach does.

Finally, Karmada [29] is a policy-based multi-cluster orchestrator for Kubernetes that enables workload propagation across heterogeneous clusters. While it supports cluster-level replication and policy-driven placement, it does not account for microservice interdependencies or perform latency- and cost-aware optimization, as our approach does. Nevertheless, both Karmada and Liqo [25] can be employed as infrastructure backends to enforce the placement decisions computed by GMA.

VIII. Conclusion

We presented a new analytical model for evaluating the delay of microservice-based applications. The model incorporates key features commonly used in cloud environments, such as Horizontal Pod Autoscalers (HPA) and locality-aware load balancing enabled by service mesh technologies.

The model guided the design of a greedy placement strategy that minimizes deployment costs while ensuring that average user delay remains below a specified threshold. The proposed strategy leverages request traces and demonstrated promising results in simulations when compared to state-of-the-art approaches.

We implemented the strategy in a novel open-source Kubernetes controller [14], designed to maintain user delay within a defined range. The proposed controller has a high level of automation and can be used with any application able to export telemetry data to the Istio service mesh. Experimental results confirmed the effectiveness of the controller in dynamically managing microservice placement to meet performance requirements.

References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[2] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[3] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[4] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 122–1225.

[5] B. Li, Q. He, G. Cui, X. Xia, F. Chen, H. Jin, and Y. Yang, "Read: Robustness-oriented edge application deployment in edge computing environment," *IEEE Transactions on Services Computing*, vol. 15, no. 3, pp. 1746–1759, 2022.

[6] S. Deng, Z. Xiang, J. Taheri, M. A. Khoshkholghi, J. Yin, A. Y. Zomaya, and S. Dustdar, "Optimal application deployment in resource constrained distributed edges," *IEEE Transactions on Mobile Computing*, vol. 20, no. 5, pp. 1907–1923, 2021.
[7] B. Costa, J. Bachiega Jr, L. R. De Carvalho, and A. P. Araujo, "Orchestration in fog computing: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 55, no. 2, pp. 1–34, 2022.
[8] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.
[9] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-aware scheduling in container-based clouds," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 4461–4477, 2023.
[10] L. Toka, "Ultra-reliable and low-latency computing in the edge with kubernetes," *Grid Computing*, vol. 19, no. 31, pp. 1572–9184, 2021.
[11] G. Li, H. Liu, H. Zeng, S. Kandula, D. A. Maltz, M. Zhang, B. Y. Zhao, and H. V. Madhyastha, "Towards automated performance diagnosis in a large cdn," in *Proceedings of ACM SIGCOMM*, 2017, pp. 39–52.
[12] N. T. Blog, "Netflix open connect," Online, 2020, available: https://netflixtechblog.com/netflix-open-connect-2016-1460f3dc3e4.
[13] A. Detti, "Microservices from cloud to edge: An analytical discussion on risks, opportunities and enablers," *IEEE Access*, vol. 11, pp. 49 924–49 942, 2023.
[14] A. Favale and A. Detti, "muPlacer," 2024. [Online]. Available: https://github.com/alessandrofavale/muPlacer
[15] A. Detti, L. Funari, and L. Petrucci, "μBench: An Open-Source Factory of Benchmark Microservice Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 968–980, 2023.
[16] F. Du, J. Shi, Q. Chen, P. Pang, L. Li, and M. Guo, "Generating microservice graphs with production characteristics for efficient resource scaling," in *Proceedings of the 2025 International Conference on Supercomputing (ICS)*. ACM, 2025, p. —.
[17] J. Shi, H. Zhang, Z. Tong, Q. Chen, K. Fu, and M. Guo, "Nodens: Enabling resource efficient and fast {QoS} recovery of dynamic microservice applications in datacenters," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 403–417.
[18] Google kubernetes engine pricing. (Accessed 2024-12-30). [Online]. Available: https://cloud.google.com/kubernetes-engine/pricing
[19] Google cloud all network pricing. (Accessed 2024-12-30). [Online]. Available: https://cloud.google.com/vpc/network-pricing
[20] L. Kleinrock, *Queueing systems: Computer Applications*. wiley, 1974, vol. 2.
[21] Jaeger tracing. (Accessed 2024-12-30). [Online]. Available: https://www.jaegertracing.io
[22] R. Karp, "Reducibility among combinatorial problems," *Complexity of Computer Computations*, vol. 40, pp. 85–103, 01 1972.
[23] A. Marchese and O. Tomarchio, "Network-aware container placement in cloud-edge kubernetes clusters," in *2022 22nd IEEE international symposium on cluster, cloud and internet computing (CCGrid)*. IEEE, 2022, pp. 859–865.
[24] Gartner, Inc., "Survey analysis: Microservices adoption trends, 2022," https://www.gartner.com/en/documents/4015423, 2022, accessed: 2025-06-25.
[25] Liqo Authors, "Liqo: Kubernetes-based multi-cluster orchestration with virtual nodes," https://liqo.io, 2025, accessed: 2025-07-14.
[26] M. Xu, Z. Fu, X. Ma, L. Zhang, Y. Li, F. Qian, S. Wang, K. Li, J. Yang, and X. Liu, "From cloud to edge: A first look at public edge platforms," in *Proceedings of Internet Measurement Conference (IMC)*, 2021.
[27] G. Martínez, J. A. Hernández, P. Reviriego, and P. Reinheimer, "Round trip time (rtt) delay in the internet: Analysis and trends," *IEEE Network*, vol. 38, no. 2, pp. 280–285, 2024.
[28] C. Attig, N. Rauh, T. Franke, and J. F. Krems, "System latency guidelines then and now – is zero latency really considered necessary?" in *Engineering Psychology and Cognitive Ergonomics 2017*, ser. Lecture Notes in Computer Science. Springer, 2017, vol. 10276, pp. 2–14.
[29] Karmada Authors, "Karmada: Open, multi-cluster kubernetes orchestration," https://karmada.io, 2025, accessed: 2025-07-14.
[30] T. Pusztai, F. Rossi, and S. Dustdar, "Pogonip: Scheduling asynchronous applications on the edge," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 660–670.
[31] S. Wang, M. Zafer, and K. K. Leung, "Online placement of multi-component applications in edge computing environments," *IEEE Access*, vol. 5, pp. 2514–2533, 2017.
[32] K. Ray, A. Banerjee, and N. C. Narendra, "Proactive microservice placement and migration for mobile edge computing," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, 2020, pp. 28–41.
[33] L. Liu, H. Tan, S. H.-C. Jiang, Z. Han, X.-Y. Li, and H. Huang, "Dependent task placement and scheduling with function configuration in edge computing," in *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, 2019, pp. 1–10.
[34] S. B. Nath, S. Chattopadhyay, R. Karmakar, S. K. Addya, S. Chakraborty, and S. K. Ghosh, "Ptc: Pick-test-choose to place containerized micro-services in iot," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.
[35] W. Lv, Q. Wang, P. Yang, Y. Ding, B. Yi, Z. Wang, and C. Lin, "Microservice deployment in edge computing based on deep q learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2968–2978, 2022.
[36] L. Wang, X. Deng, J. Gui, X. Chen, and S. Wan, "Microservice-oriented service placement for mobile edge computing in sustainable internet of vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 9, pp. 10 012–10 026, 2023.
[37] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016, pp. 20–26.

Andrea Detti is a professor of Wireless Networks and Cloud Computing at the University of Rome "Tor Vergata". His current research activity spans on different topics in the area of computer networks and cloud computing. He is co-author of more than 80 papers on journals and conference proceedings, and participated to several EU funded projects with coordination and research roles.

Alessandro Favale graduated with a master's degree in ICT and Internet Engineering from the University of Rome Tor Vergata. His interests range from cloud computing to microservices applications and networking. His current work focuses on cloud-native architectures, Kubernetes orchestration, enterprise networking and storage solutions.