

## RESEARCH ARTICLE

# Microservices From Cloud to Edge: An Analytical Discussion on Risks, Opportunities and Enablers

**ANDREA DETTI** , (Member, IEEE)Department of Electronic Engineering, University of Rome "Tor Vergata," 00133 Rome, Italy  
National Inter-University Consortium for Telecommunications (CNT), 43124 Parma, Italy

e-mail: andrea.detti@uniroma2.it

This work was supported in part by the Italian RESTART Program funded by European Union–NextGenerationEU through Italian PNRR (Piano Nazionale di Ripresa e Resilienza).

**ABSTRACT** Microservice applications are made of many interacting microservices that can leverage different cloud servers to distribute the computational load horizontally, so these applications are considered cloud-native. The emergence of edge computing has prompted the research of placement strategies that consider the possibility of instantiating microservices in cloud and edge data centers to improve specific performance metrics, such as the time the application spent serving a user request, also referred to as user delay or makespan. Microservice applications differ in many architectural properties, such as the number of microservices, the number of them involved per request, their dependency graph, and the presence of centralized databases. We found that these properties influence the possibility of a placement strategy to exploit edge resources to reduce user delay; therefore, they constitute an application-level optimization space that, however, has not yet been explored in the literature. Accordingly, this paper contributes to filling this knowledge gap by answering the following question: which are the architectural properties of a microservice application that enable it to be more *edge-native*, that is, to better reduce user delay by using edge computing? Our results are based on a new analytical modeling of the average user delay provided by a microservice application deployed on cloud/edge resources and a new heuristic placement strategy that take into account key aspects of the application, and the supporting cloud and network environment, not considered by previous literature works.

**INDEX TERMS** Edge computing, microservices, placement problems, delay performance, analytical modeling.


## I. INTRODUCTION

Microservice architecture is a widely used architectural style for enterprise software that decomposes large applications into a set of small, modular, and independently deployable “micro” services. To serve user requests, each microservice performs a specific internal function and communicates with other microservices through network APIs, usually based on HTTP or gRPC [1], [2], [3].

The benefits of moving from a monolithic architecture to a microservices architecture are many and involve both development and performance aspects. Decoupling the application into small loosely coupled components makes it easier to understand and program, thus improving developer

productivity and accelerating release cycles. It is possible to horizontally distribute the application workload on a cluster of servers rather than using an expensive giant server and manage resource allocation precisely by replicating or giving more resources to the most heavily loaded microservices or those that require more reliability. For these characteristics, microservice applications are cloud-native i.e., able to run and scale in modern, dynamic environments such as public, private and hybrid clouds. The disadvantages of decomposing an application into microservices include increased latency and processing load due to internal network interactions, increased difficulty in debugging, etc. However, the benefits are more significant, especially for complex applications that need to support high request loads [4].

The most widely used technology today to package microservices are Linux Containers, managed by Docker [5]

The associate editor coordinating the review of this manuscript and approving it for publication was Rodrigo S. Couto .

or other Container engines. Then automating deployment, scaling, and management of Containers (microservices) on a cluster of real or virtual servers is usually done using the Kubernetes (k8s) Container Orchestration platform [6], possibly supported by other frameworks, for example, the so-called service meshes for observability and request routing [7].

Along with the growing adoption of microservice architectures, edge computing is another architectural paradigm in which there is great interest in bringing computation and data storage closer to the devices that generate and/or consume the data. This is in contrast to traditional cloud computing, which centralizes computation and storage in remote data centers. When microservices are deployed on the edge, they enable users to process their requests closer, which can offer many benefits, such as reduced latency and network traffic toward central data centers, which can lead to cost savings and more efficient use of network resources [8], [9].

Edge resources are expected to be less and more expensive than cloud resources. For example, the physical space where to deploy servers is smaller at the edge than in a central data center and the resource utilization efficiency in a central cloud is higher because there is a larger aggregation of request flows. This motivated research on solutions that optimize the use of precious edge resources and, consequently, microservice placement strategies for edge computing scenarios have been extensively addressed in the literature with the goal of understanding which microservices are best run in the edge and which in the cloud data center to improve a specific performance metrics, such as the time the application spent serving a user request, also referred to as user delay or makespan [10], [11], [12].

### A. RESEARCH OBJECTIVES

Microservice applications differ in many architectural properties, such as the number of microservices, the number of them involved per request, their dependency graph, the presence of centralized databases, and the type of interactions (e.g., request-response or publish-subscribe). We found that depending on these characteristics, some applications are more *edge-native* than others; i.e., placement strategies can potentially reduce their user delay more significantly by exploiting edge resources. The term “potentially” means that the reduction of delays could not be realized due to inefficiencies of the placement strategy employed, but the reduction of delays occurs by using an optimal placement strategy.

This finding led us to argue that the reduction of the delay of a microservice application by exploiting edge computing can be addressed on the one hand by designing optimal or near-optimal placement strategies and, on the other hand, by designing the application itself so that it has those architectural properties that allow the placement strategy to be most effective in reducing delay using edge resources. To the best of our knowledge, this latter application-level optimization

domain has not yet been explored in the literature. Accordingly, this paper contributes to filling this knowledge gap by analyzing how different application properties influence the potential improvement that edge computing can provide in terms of reducing user delay.

### B. METHODOLOGY

To perform the analysis, we developed a novel analytical model of the average user delay of a microservice application that takes as input i) the properties of a microservice application, ii) the placement state of the microservices, i.e., which microservice is running in the cloud and which replica is also running in the edge, iii) the rate of user requests, and also iv) key aspects of the supporting execution environment, such as the routing policy used by the service mesh to route requests among microservice instances, the amount of CPU and network resources to be shared based on processor sharing paradigms.

The metric we consider to compare applications with different properties is the average user delay provided by the analytical model in the case of optimal (or near-optimal) placement solutions, that is, solutions that minimize the delay of the considered applications. In so doing, the differences in delay among the compared applications are only due to their different architectural properties and are not “biased” by possible inefficiencies of the placement strategy.

To find an optimal placement solution for the microservices of an application, we tried to use an exhaustive searching approach that merely evaluates the user delay for all possible placement solutions and chooses the one that provides the minimum delay. Unfortunately, the search space is already so large for applications with a few microservices that exhaustive search is impractical, and the minimization problem is also NP-complete. As a result, we had to resort to near-optimal placement solutions, based on a new heuristic strategy we developed as a consequence of the fact that those in the literature did not include all application properties and system characteristics of our interest, as we will comment by discussing related work.

### C. CONTRIBUTIONS

Overall, the specific contributions of the paper presented in the next sections are as follows.

- In Sec. II, we propose an analytical model of the average user delay of microservice applications deployed in edge and cloud data centers that takes into account application and system properties not considered by previous models.
- In Sec. III, we formulate an edge/cloud microservice placement problem whose solutions are optimal in the sense of minimizing user delay.
- In Sec. IV, we provide an understanding of the characteristics that an optimal placement solution has, and this allowed us to develop a suboptimal heuristic algorithm, called Path Adding Microservice Placement (PAMP).

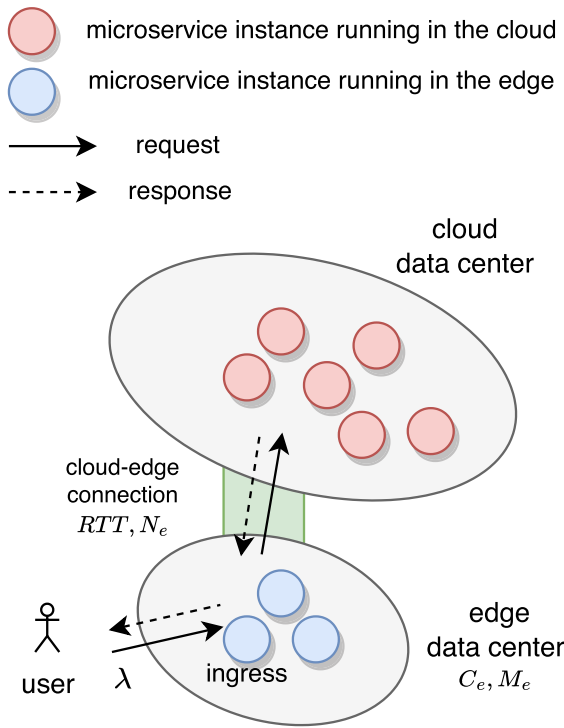


FIGURE 1. Reference scenario.

- In Sec. V, we use the analytical model of user delay combined with the placement solutions provided by the PAMP algorithm to analyze which architectural properties of the microservice application are enablers of edge computing, i.e., they allow a better reduction of user delays by using edge resources.<sup>1</sup>
- In Sec. VI we comment on the differences between our work and that of the literature, and finally in Sec. VII we draw conclusions.

## II. ANALYTICAL MODEL OF THE USER DELAY

### A. SYSTEM MODEL

#### 1) INFRASTRUCTURE

We consider a generic system consisting of edge data centers and a remote cloud data center. We assume that there are no edge-to-edge interactions, i.e., service requests are processed either in the origin edge data center or in the cloud. Consequently, we restrict our analysis to the case of a single edge data center, as shown in Fig. 1. Data centers have processing (CPU), memory, and network resources. Cloud data center resources are assumed to be unlimited. The edge data centers have a total number of CPUs equal to  $C_e$  and memory equal to  $M_e$ . There is a bidirectional network connection between

<sup>1</sup>In the rest of the paper, we will often briefly say that a given architectural property allows better or worse exploitation of edge computing. More formally, we mean that the architectural property under consideration potentially allows placement strategies to more or less reduce user delay by exploiting edge resources.

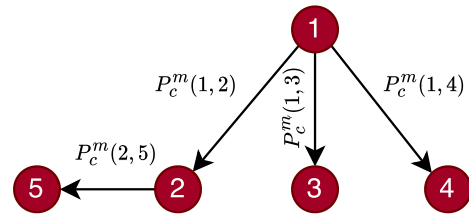


FIGURE 2. Dependency graph of microservices ( $G_m$ ).

the cloud and the edge, whose capacity is equal to  $N_e$  in each direction. The data centers' resources are used to deploy (i.e., execute) the instances of the microservices of the application, where with the term instance we mean a running version of the microservice code.

#### 2) USER

As shown in Fig. 1, the user is located close to the edge data center and sends requests to a “ingress” microservices of an application, that is, those microservices that the user can directly contact. We do not consider the possible delay introduced by the user-edge connection, also because it is a parameter that cannot be optimized in our system. Service requests are generated at a rate of  $\lambda$  requests per second and are always routed to the edge data center. Instances of microservices running in the edge can serve the request independently or interact with other microservices whose instances are running in the cloud.

#### 3) MICROSERVICE APPLICATION

The microservice application consists of  $M - 1$  microservices that interact using a request-response model (e.g., REST HTTP API) and whose work model follows the one proposed in [4]. Specifically, when a microservice  $i$  receives a request, it executes a *internal function* and then sequentially calls a set of downstream microservices according to a *microservice-level* call probability  $P_c^m$ . Specifically, the downstream microservice  $j$  is called with probability  $P_c^m(i, j)$ . When all called microservices send their response back, the microservice  $i$  sends its response to the upstream caller and terminates the processing of the request. We consider the user as a  $M$ th microservice and  $P_c^m(M, j)$  is the probability that the user sends his request to the microservice  $j$ . The matrix  $P_c^m$  can be used to build the dependency graph  $G_m$  of microservices that has a node per microservice and a link between  $i$  and  $j$  if  $P_c^m(i, j) > 0$ . We assume that  $G_m$  is a directed acyclic graph (DAG). Fig. 2 shows an example of a dependency graph for a microservice application made of 5 microservices, whose ingress microservice is the number 1. When microservice 1 receives a request from the user, it can call microservices 2,3 and 4 to serve a request with probability  $P_c^m(1, 2)$ ,  $P_c^m(1, 3)$ ,  $P_c^m(1, 4)$ , respectively. In turn, if microservice 2 is called, then it can call microservice 5 with probability  $P_c^m(2, 5)$ .

TABLE 1. Major notation.

Notation	Default value	Description
$C_e, M_e$	8 CPUs, 4C <sub>e</sub> GB	Total CPUs and memory of the edge data center.
$N_e, RTT$	400 Mbit/s, 30ms	Bitrate ( $N_e$ ) and round trip time ( $RTT$ ) of the bidirectional connection between cloud and edge data centers (bit/sec).
$\lambda$	40 req/s	Average number of requests per second sent by the user.
$M$	31	$M - 1$ is the number of microservices of the application.
$i, j$		Identifiers of microservices.
$d_i, d_j$		Identifiers of data centers. The values can be "edge" or "cloud".
$\hat{I}, \langle i, d_i \rangle, \hat{J}, \langle j, d_j \rangle$		Identifier of an instance of the microservice $i$ that runs in the data center $d_i$ ( $\hat{I}$ or $\langle i, d_i \rangle$ ) and an instance of the microservice $j$ that runs in the data center $d_j$ ( $\hat{J}$ or $\langle j, d_j \rangle$ ).
$\hat{U}$ or $\langle M, \text{edge} \rangle$		Microservice instance used to identify the user.
$R_{cpu\_srr}(\hat{I})$	Random in 1+20 ms	CPU seconds needed per request to process the internal function of instances of microservice $i$ . Default values per microservice $i$ are randomly chosen in the range of 1+20 ms, and different instances of the same microservice have the same values.
$R_{cpu}(\hat{I})$	100 R <sub>cpu_srr</sub> ( $\hat{I}$ )	CPU quota dedicated to instances of microservice $i$ .
$R_{mem}(\hat{I})$	250 MB	Amount of memory (bytes) dedicated to instances of microservice $i$ .
$R_s(\hat{I})$	800 bits	Size of the response generated by the instances of microservice $i$ .
$R_{cpu}^{tot}, R_{mem}^{tot}$		Total CPUs and memory consumed on the edge data center
$m_{xc}$	9	Average number of involved microservice per request.
$P_c^m(i, j)$	$p \mid m_{xc} = 9$	Probability that the microservice $i$ calls the microservice $j$ during the processing of a request. For the default configuration, it is equal to a constant $p$ that ensures the chosen value of $m_{xc}$ .
$P_c^i(\hat{I}, \hat{J})$		Probability that microservice instance $\hat{I}$ calls microservice instance $\hat{J}$ during the processing of a request.
$G_m, G_i$		Dependency graphs of microservices and microservice instances, respectively.
$\Pi, \pi$		$\Pi$ Set of all paths of $G_m$ from the user to any microservice. $\pi$ single generic path of $\Pi$ called dependency path.
$S(\hat{I})$		$S(\hat{I}) = 1$ means that there exists a microservice instance $\hat{I}$ , otherwise $S(\hat{I})=0$ .
$N_c(\hat{I})$		Average number of times the microservice instance $\hat{I}$ is called during the processing of a request.
$D_m(\hat{I})$		Average execution time of the microservice instance $\hat{I}$ .
$D_m(\hat{U})$		Average execution time of a user request.
$D_i(\hat{I})$		Average execution time of the internal function of the microservice instance $\hat{I}$ .
$\rho_c(\hat{I}), \rho_{nce}, \rho_{nec}$		Utilization factors of: the CPU resources of microservice instances $\hat{I}$ ( $\rho_c(\hat{I})$ ), cloud-edge network connection ( $\rho_{nce}$ ), edge-cloud network connection ( $\rho_{nec}$ ).
$D_n(\hat{I}, \hat{J}), P_d, Td(\hat{I}, \hat{J})$		$D_n(\hat{I}, \hat{J})$ is the network time needed to transfer a request from the microservice instance $\hat{I}$ to the microservice instance $\hat{J}$ and receive the response. It accounts for propagation $P_d$ and data transmission $Td(\hat{I}, \hat{J})$ delays.
$T_{nce}, T_{nec}$		Network traffic volume (bit/s) on cloud-edge and edge-cloud connection, respectively.
$\eta_e$		Edge distribution factor equal to the ratio of microservices running in the edge to the total number of microservices of the application.
$\rho_e$		Edge utilization factor equal to the number of CPUs used by edge microservices and the total number of edge CPUs.
$N_{RT}$		Average number of times microservices instances in the edge need to interact with those in the cloud to serve a request or vice versa.
$\alpha, a$	random selection of a configuration in Table 2	B-A algorithm parameters. $\alpha$ is the power of preferential attachment, and $a$ is the attractiveness of nodes without children.

Each microservice has an *instance* that runs in the cloud data center. Furthermore, edge resources can be used opportunistically to run another instance of the microservice in the edge data center to process requests closer to the user.<sup>2</sup> We identify a microservice instance with a tuple  $\langle i, d_i \rangle$ . The index  $i$  identifies the microservice, the index  $d_i$  is a string that identifies the data center where the instance is running, specifically,  $d_i = \text{cloud}$  for an instance running in the cloud

<sup>2</sup>With the term microservice, we simply refer to the software code of the microservice. A running version of the code is called a microservice *instance*. In the presence of many instances of the same microservice in the same data center, a microservice instance of our model represents the whole replica set.

and  $d_i = \text{edge}$  for an instance running at the edge. To simplify the notation, we represent a tuple with a single letter, specifically  $\hat{I} = \langle i, d_i \rangle$  and  $\hat{J} = \langle j, d_j \rangle$ . The user is the microservice instance  $\hat{U} = \langle M, \text{edge} \rangle$ . For example, Fig. 3 shows a possible edge/cloud placement of instances of the microservices of the application in Fig. 2. All microservices have a cloud instance, and microservices 1,3 and 4 also have an edge instance.

The state of the application represents where its instances are running and is a binary placement vector  $S$  that has an element for each possible microservice instance  $\hat{I}$ , therefore, it has  $2M$  elements since each microservice instance can run

- i,c instance  $\langle i, \text{cloud} \rangle$  of microservice  $i$  running in the cloud
- i,e instance  $\langle i, \text{edge} \rangle$  of microservice  $i$  running in the edge

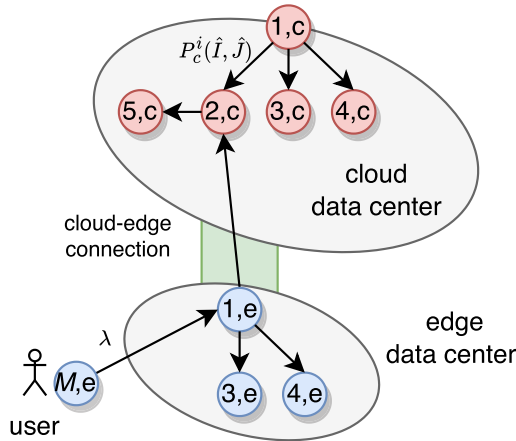


FIGURE 3. Dependency graph of microservices instances ( $G_i$ ).

in both cloud and edge data centers. A value  $S(\hat{I}) = 1$  means that the microservice instance  $\hat{I}$  is running. Vice versa, for  $S(\hat{I}) = 0$ , the microservice instance  $\hat{I}$  does not exist. Since we assume that an instance of a microservice always runs in the cloud, we have  $S(\hat{I}) = 1$ , for  $\hat{I} = \langle i, \text{cloud} \rangle$  with  $1 \leq i \leq M - 1$ . The user is only on the edge, therefore  $S(\hat{U}) = 1$  and  $S(\hat{I}) = 0$  for  $\hat{I} = \langle M, \text{cloud} \rangle$ . Other values of  $S$  are those that the placement strategy aims to optimize.<sup>3</sup>

Regarding the resource consumption of a microservice instance  $\hat{I}$ , we assume that only the execution of the internal function consumes computational resources. The CPU seconds needed per request to process the internal function is equal to  $R_{cpu\_srx}(\hat{I})$ .<sup>4</sup> The CPU quota and the amount of memory dedicated are equal to  $R_{cpu}(\hat{I})$  and  $R_{mem}(\hat{I})$ , respectively.<sup>5</sup> The size of the generated response is  $R_s(\hat{I})$  bits; the size of the requests is negligible. For the user, we simply set  $R_{cpu\_srx}(\hat{U}) = R_{cpu}(\hat{U}) = R_{mem}(\hat{U}) = R_s(\hat{U}) = 0$

#### 4) SERVICE MESH

The service mesh is a management framework that controls the routing of requests towards different instances of a microservice to comply with a routing policy [7], [13], [14], [15]. The topology of the service mesh is a dependency graph  $G_i$  of microservice instances (Fig. 3). The nodes of  $G_i$  are instances and there is a link between node  $\hat{I}$  and  $\hat{J}$

<sup>3</sup>The size of  $S$  and later variables could be reduced by taking these constraints into account. However, we prefer to keep this general mathematical framework for future analyses where these constraints could be relaxed.

<sup>4</sup>Note that a CPU is shared among processes, therefore  $R_{cpu\_srx}(\hat{I})$  would be the execution time of the internal function only in case of full and unshared access to a CPU.

<sup>5</sup>The CPU quota is a number greater than zero and indicates, for each second, how many CPU seconds are dedicated to a microservice instance. For example,  $R_{cpu}(\hat{I}) = 0.2$  (aka 200 millicpu) means that on average the instance can use 20% of a CPU,  $R_{cpu}(\hat{I}) = 2$  means that an instance can use 2 CPUs, and so on.

if instance  $\hat{I}$  can call instance  $\hat{J}$  while processing a request. We statistically model this event with a *instance-level* call probability named  $P_c^i(\hat{I}, \hat{J})$ .

The values  $P_c^i$  are correlated with the microservice-level call probabilities  $P_c^m$ , and this correlation depends on the routing policy. We assume a typical *locality load balancing* policy, for which the interactions between microservices are preferentially resolved locally [14], in the same data center. This policy can be modeled by configuring the call probabilities  $P_c^i$  as shown in algorithm 1, where it is convenient to make explicit the tuples  $\hat{I} = \langle i, d_i \rangle$  and  $\hat{J} = \langle j, d_j \rangle$  to identify microservice instances.

To summarize, assuming that a microservice  $i$  calls a microservice  $j$  with probability  $P_c^m(i, j) > 0$  and that therefore there exists a  $i, j$  link in  $G_m$ , then

- the instance of microservice  $i$  running in the cloud calls the instance of microservice  $j$  running in the cloud with probability  $P_c^i(\langle i, \text{cloud} \rangle, \langle j, \text{cloud} \rangle) = P_c^m(i, j)$  (line 5), so there exists a  $G_i$  link between  $\langle i, \text{cloud} \rangle$  and  $\langle j, \text{cloud} \rangle$ ;
- an instance of the microservice  $i$  running in the edge calls the instance of the microservice  $j$  running in the edge, if it exists, with probability  $P_c^i(\langle i, \text{edge} \rangle, \langle j, \text{edge} \rangle) = P_c^m(i, j)$  (line 9) and thus there exists a link in  $G_i$  between  $\langle i, \text{edge} \rangle$  and  $\langle j, \text{edge} \rangle$ ; otherwise, the instance of microservice  $i$  running in the edge calls the instance of the microservice  $j$  running in the cloud (line 11) with probability  $P_c^i(\langle i, \text{edge} \rangle, \langle j, \text{cloud} \rangle) = P_c^m(i, j)$  and thus there exists a link in  $G_i$  between  $\langle i, \text{edge} \rangle$  and  $\langle j, \text{cloud} \rangle$ .

For example, Fig. 3 shows the instance dependency graph  $G_i$  for the microservice application whose microservice dependency graph  $G_m$  is shown in Fig. 2 and when microservices 1,3 and 4 have an instance running in the edge.

We note that the model we propose is suitable for implementing other routing policies by appropriately modifying the algorithm 1.

---

#### Algorithm 1 $P_c^i$ for Locality Load Balancing Routing Policy

---

- 1: Initialize  $P_c^i$  with 0 values
  - 2: **for**  $i = 1$  to  $M$  **do**
  - 3:     **for**  $j = 1$  to  $M$  **do**
  - 4:         **if**  $i \neq M$  **then** ▷ not for the user
  - 5:              $P_c^i(\langle i, \text{cloud} \rangle, \langle j, \text{cloud} \rangle) = P_c^m(i, j)$
  - 6:         **end if**
  - 7:         **if**  $S(\langle i, \text{edge} \rangle) = 1$  **then**
  - 8:             **if**  $S(\langle j, \text{edge} \rangle) = 1$  **then**
  - 9:                  $P_c^i(\langle i, \text{edge} \rangle, \langle j, \text{edge} \rangle) = P_c^m(i, j)$
  - 10:             **else**
  - 11:                  $P_c^i(\langle i, \text{edge} \rangle, \langle j, \text{cloud} \rangle) = P_c^m(i, j)$
  - 12:             **end if**
  - 13:         **end if**
  - 14:     **end for**
  - 15: **end for**
- 

We conclude the section by presenting, in Fig. 4, an example of the interactions that occur among microservices to

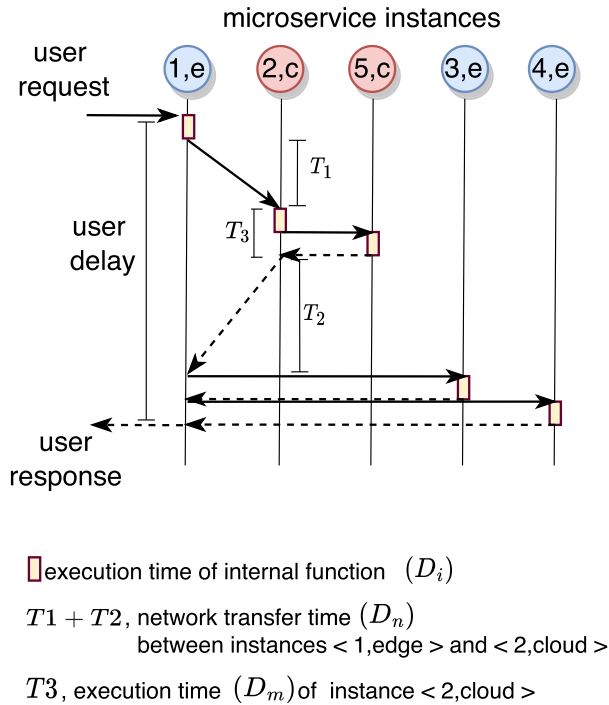


FIGURE 4. Trace of a user request.

serve a user request in the case of the configuration in Fig. 3. The user's request is received by the instance (1, edge) of microservice 1 running in the edge data center, which performs its internal function and then calls microservice 2. The service mesh routes this request to the instance (2, cloud) of microservice 2 running in the cloud, so the request traverses the edge-to-cloud network connection. The instance (2, cloud) executes its internal function and then calls microservice 5. This request is routed by the service mesh to the local instance (5, cloud). After the execution of its internal function, instance (5, cloud) sends its response to instance (2, cloud) which in turn sends its response to edge instance (1, edge). This response traverses the cloud-to-edge network connection. As a result, the microservice instance (1, edge) calls microservices 3 and 4, these calls are routed by the service mesh to the related local edge instances, and when, finally, the instance (4, edge) responds to (1, edge), it responds to the user.

**B. DELAY MODEL**

Now that we have modeled the reference system, we move on to analytically evaluate the average delay of user requests (Fig. 4), as a function of the state  $S$  of the application. This model will be used in the next section to formulate the delay minimization problem.

We define  $D_m(\hat{I})$  as the execution time of the microservice  $\hat{I}$  that is equal to the sum of:

- 1) the time  $D_i(\hat{I})$  required by the microservice  $\hat{I}$  to execute its internal function,

- 2) the execution times  $D_m(\hat{J})$  of each downstream microservice  $\hat{J}$ ,
- 3) the network times  $D_n(\hat{I}, \hat{J})$  required to transfer requests and responses between microservices  $\hat{I}$  and  $\hat{J}$ .

The last two values are weighted by the call probabilities  $P_c^i(\hat{I}, \hat{J})$ .

For a microservice  $\hat{I}$  that is not running ( $S(\hat{I}) = 0$ ), we used the dummy value of 0 for  $D_m(\hat{I})$  and for related other values as well. Accordingly, to simplify the notation, in the next formulas, we will avoid making explicit that they refer to values for which all considered microservices are running; otherwise the value is zero. Consequently, we can write the following equation.

$$D_m(\hat{I}) = D_i(\hat{I}) + \sum_j P_c^i(\hat{I}, \hat{J}) (D_m(\hat{J}) + D_n(\hat{I}, \hat{J})) \quad (1)$$

Considering all possible values of  $\hat{I}$ , we have a system of equations whose resolution returns the values  $D_m(\hat{I})$ ; among them, the value  $D_m(\hat{U})$  is the time it takes the user to execute a request, so it is the average delay of the requests that we intend to minimize.<sup>6</sup>

The execution time  $D_i(\hat{I})$  of the internal function is not a constant but depends on the computing resources dedicated to microservice instance  $\hat{I}$  and a load of requests that the instance receives. Indeed, we consider that an instance serves more than one request. To compute  $D_i(\hat{I})$ , we modeled the use of computing resources as an M/G/1 queue with *processor-sharing*.<sup>7</sup> Accordingly,  $D_i(\hat{I})$  can be expressed as [16]:

$$D_i(\hat{I}) = \begin{cases} \frac{R_{cpu\_srx}(\hat{I})/R_{cpu}(\hat{I})}{1 - \rho_c(\hat{I})}, & \text{for } \hat{I} \neq \hat{U} \\ 0, & \text{for } \hat{I} = \hat{U} \end{cases} \quad (2)$$

$\rho_c(\hat{I})$  is the utilization factor of the CPU quota dedicated to microservice instance  $\hat{I}$ , which is equal to

$$\rho_c(\hat{I}) = \begin{cases} \min\left(\frac{\lambda N_c(\hat{I}) R_{cpu\_srx}(\hat{I})}{R_{cpu}(\hat{I})}, 1\right), & \text{for } \hat{I} \neq \hat{U} \\ 0, & \text{for } \hat{I} = \hat{U} \end{cases} \quad (3)$$

$N_c(\hat{I})$  is the average number of times in which microservice instance  $\hat{I}$  is called during the processing of a user request. This value depends on the calling probabilities  $P_c^i$  as follows

$$N_c(\hat{I}) = \begin{cases} \sum_j N_c(\hat{J}) P_c^i(\hat{J}, \hat{I}), & \text{for } \hat{I} \neq \hat{U} \\ 1, & \text{for } \hat{I} = \hat{U} \end{cases} \quad (4)$$

<sup>6</sup>Since the dependency graph of microservices  $G_m$  is a DAG, the dependency graph of instances  $G_i$  is also a DAG, so a topological ordering of nodes can be derived and this system can be solved iteratively with  $2M$  steps. This also applies to the following systems of equations.

<sup>7</sup>A microservice instance usually processes multiple requests in parallel, which share CPU resources dedicated to the instance. This justifies the processor-sharing model. The assumption of an exponential interarrival time is a simplification to address the problem analytically. Although numerical results are not accurate in the absence of this type of interarrival, the conclusions we are seeking about the impact of system parameters on edge computing effectiveness are still valid.

The first equation states that the number of times a microservice instance  $\hat{I}$  is involved in a request is equal to the number of times each upstream microservice instance  $\hat{J}$  is involved, multiplied by the calling probability  $P_c^i(\hat{J}, \hat{I})$ . The last equation takes into account that the user is always involved in a request. Considering all possible values of  $\hat{I}$ , we have a system of equations from which the values of  $N_c(\hat{I})$  can be derived.

For network time  $D_n(\hat{I}, \hat{J})$ , we assume that if microservice instances  $\hat{I}$  and  $\hat{J}$  are placed in the same data center, this delay is negligible, because we are considering a low-latency and very high-speed network within a data center. When  $\hat{I}$  and  $\hat{J}$  are in different data centers, the network time  $D_n(\hat{I}, \hat{J})$  is equal to a constant  $P_d$  that depends on the propagation delay between data centers, plus the time  $T_d(\hat{I}, \hat{J})$  needed to transmit  $R_s(\hat{J})$  bits from instance  $\hat{J}$  to  $\hat{I}$  over the cloud-edge connection. For example, for HTTP REST APIs,  $P_d$  can be approximated to  $3RTT$ , where  $RTT$  is the round trip time between data centers. Two RTTs are needed for opening and closing the TCP connection and another RTT takes into account the propagation of the request and the propagation of the response. In formula,

$$D_n(\hat{I}, \hat{J}) = \begin{cases} P_d + T_d(\hat{I}, \hat{J}), & \text{if } d_i \neq d_j \\ 0, & \text{otherwise.} \end{cases} \quad (5)$$

for which, we recall that  $d_i$  and  $d_j$  identify the data centers of microservice instances  $\hat{I}$  and  $\hat{J}$ .

The transmission time  $T_d(\hat{I}, \hat{J})$  is not a constant because it depends on the capacity of the cloud-edge connection and the traffic load. Accordingly, we model cloud-edge network resource sharing as two M/G/1 queues with processor sharing, one per direction.<sup>8</sup> It follows that time  $T_d(\hat{I}, \hat{J})$  can be written as

$$T_d(\hat{I}, \hat{J}) = \begin{cases} \frac{R_s(\hat{J})/N_e}{1 - \rho_{nce}}, & \text{for } d_i = \text{edge}, d_j = \text{cloud} \\ \frac{R_s(\hat{J})/N_e}{1 - \rho_{nec}}, & \text{for } d_i = \text{cloud}, d_j = \text{edge} \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

The values  $\rho_{nce}$  and  $\rho_{nec}$  are the utilization factors of the cloud-to-edge and edge-to-cloud connections, respectively. They can be written as,

$$\rho_{nce} = \min(T_{nce}/N_e, 1) \quad (7)$$

$$\rho_{nec} = \min(T_{nec}/N_e, 1) \quad (8)$$

where  $T_{nce}$  and  $T_{nec}$  are the volumes of cloud-to-edge and edge-to-cloud traffic, respectively, and can be evaluated as

<sup>8</sup>We used processor sharing to model the fact that concurrent TCP/IP connections share transmission resources fairly. Exponential interarrivals among TCP/IP connections, on the other hand, is a simplifying assumption. However, we believe that the conclusions we draw about the effectiveness of edge computing are not compromised in the absence of such an interarrival distribution.

follows.

$$T_{nce} = \sum_{\hat{I}, d_i=\text{edge}} \lambda N_c(\hat{I}) \sum_{\hat{J}, d_j=\text{cloud}} P_c^i(\hat{I}, \hat{J}) R_s(\hat{J}) \quad (9)$$

$$T_{nec} = \sum_{\hat{I}, d_i=\text{cloud}} \lambda N_c(\hat{I}) \sum_{\hat{J}, d_j=\text{edge}} P_c^i(\hat{I}, \hat{J}) R_s(\hat{J}) \quad (10)$$

$$(11)$$

We conclude the section by computing the value of the average number of times microservices instances in the edge need to contact those in the cloud or vice versa. We simply name this value as the average number of *edge/cloud interactions*  $N_{RT}$ , it will be used to comment on some results of the analysis and can be expressed as follows.

$$N_{RT} = \sum_{\hat{I}, d_i=\text{edge}} N_c(\hat{I}) \sum_{\hat{J}, d_j=\text{cloud}} P_c^i(\hat{I}, \hat{J}) + \sum_{\hat{I}, d_i=\text{cloud}} N_c(\hat{I}) \sum_{\hat{J}, d_j=\text{edge}} P_c^i(\hat{I}, \hat{J}) \quad (12)$$

### III. PLACEMENT PROBLEM

Now we have derived all formulas to compute the average request delay  $D_m(\hat{U})$  and can formulate the placement problem to minimize the average user delay as

$$\arg \min_S D_m(\hat{U}) + \tau \quad (13)$$

$$\tau = \epsilon \left( R_{cpu}^{tot}/C_e + R_{mem}^{tot}/M_e \right) \quad (14)$$

$$R_{cpu}^{tot} = \sum_{\hat{I}, d_i=\text{edge}} S(\hat{I}) R_{cpu}(\hat{I}) \quad (15)$$

$$R_{mem}^{tot} = \sum_{\hat{I}, d_i=\text{edge}} S(\hat{I}) R_{mem}(\hat{I}) \quad (16)$$

$$\text{subject to: } R_{cpu}^{tot} \leq C_e, \quad R_{mem}^{tot} \leq M_e \quad (17)$$

$$S(\hat{U}) = 1 \quad (18)$$

$$S(\hat{I}) = 1 \quad \forall i, d_i = \text{cloud} \quad (19)$$

$$S(\hat{I}) = 0 \quad \forall i \in \Gamma, d_i = \text{edge} \quad (20)$$

$R_{cpu}^{tot}$  and  $R_{mem}^{tot}$  are the total CPU and memory resources required for the  $S$  solution. The cost  $\tau$  combined with a very small value of  $\epsilon$  avoids the unnecessary use of edge resources when, practically, the same delay can be achieved with only cloud resources. For example, if we use  $\epsilon = 10^{-6}$  the minimization will select among the solutions that offer a minimum delay at less than 2 microseconds the one that uses fewer edge resources. As for the constraints, the first takes into account the CPU and memory limits of the edge data center.<sup>9</sup> The second constraint requires the user to be on the edge. The third constraint requires each microservice to have an instance in the cloud. The last constraint takes into

<sup>9</sup>Other resource constraints can be added, for simplicity we have considered only CPU and memory limits because they are the only ones taken into account by Kubernetes.

account the possibility that a set of  $\Gamma$  microservices cannot be replicated outside the cloud. For example, this could be the case for databases used by microservices in different edge data centers that must remain centralized to ensure high data consistency.

In the next sections, we call *optimal strategy* the one that solves the placement problem we presented. We were not able to find an efficient algorithm to solve the minimization, also because the problem is NP-complete as we will comment on in the next Sec. IV. Consequently, we resort to implementing the optimal strategy with a mere exhaustive search over all possible placement solutions.

#### IV. INTUITIONS AND PAMP HEURISTIC ALGORITHM

In this section, we propose a heuristic algorithm to compute a suboptimal placement  $S$ . We had to devise this algorithm because the complexity of an exhaustive search for the optimal placement solution prevented us from performing the analysis for medium/large-scale microservice applications. The design of the algorithm is driven by some intuitions on the optimal solution that we present. Next, we describe the algorithm and show that its performance is close enough to the optimal solution. This makes us confident that we can use it to draw general conclusions about the impact that application properties have on the exploitability of resources at the edge for delay reduction. In fact, we do not run the risk that our conclusions are flawed by serious inefficiencies in the placement algorithm used to derive them.

##### A. INTUITIONS ON OPTIMAL SOLUTIONS

Suppose that we have an application consisting of 3 microservices, whose dependency graph is a simple chain  $1 \rightarrow 2 \rightarrow 3$ . The user sends his requests to microservice 1. In an initial configuration, all microservices run in the cloud. Suppose at this point that we also run an instance of microservice 2 in the edge. This placement decision is completely ineffective in terms of performance improvement and unnecessarily reserves the edge resources. In fact, the user calls microservice 1, whose instance is in the cloud. The cloud instance of microservice 1 calls the cloud instance of microservice 2 to comply with the locality load balancing policy. As a result, the edge instance of microservice 2 will never be called. It follows that,

**Necessary Condition 1** - Assuming a locality load balancing routing policy, if in an optimal placement the microservice  $i$  runs in the edge data center, then at least one of its upstreams also runs in the edge data center.

**Proof** - If necessary condition 1 is not respected, the execution of the microservice  $i$  in the edge would consume edge resources without reducing the delay, because it would never be called due to locality load balancing. This occurrence would be suboptimal with respect to a state without microservice  $i$  running in the edge because of the  $\tau$  cost in Eq. 13.

##### B. PAMP ALGORITHM

The necessary condition 1 makes it possible to reduce the number of states to be explored to find the optimal one and allows the placement problem to be turned into a knapsack problem as follows. We call *dependency path* of the microservice  $i$  a set of microservices of the dependency graph  $G_m$  that connect the user to  $i$ . For example, in Fig. 3, the microservice 5 has a single dependency path made of microservices  $\{1,2,5\}$ . The union of all the dependency paths of each microservice forms a set that we call  $\Pi$ .<sup>10</sup>

If the optimal placement  $S$  executes the microservice  $i$  in the edge, the necessary condition 1 implies that the microservices of at least a dependency path of  $i$  are running in the edge. For example, in Fig. 1, if optimal placement  $S$  executes microservice 5 in the edge, then microservices 2 and 1 must also be executed in the edge. Consequently, the search for the optimal placement can also be done by exhaustively exploring all possible combinations of dependency paths and selecting the combination that provides the lowest latency and meets the constraints. This resembles an NP-complete knapsack problem, where the items are the dependency paths  $\pi \in \Pi$ . Calling  $S$  the new state we have by running the microservices of dependency path  $\pi$  in the edge data center, the value  $v(\pi)$  of a path is the inverse of the request delay  $D_m(\hat{U})$  we get with  $S$ , and the weight  $w(\pi)$  is the amount of edge resources consumed by state  $S$ .

Consequently, we can address the problem with a typical greedy *suboptimal* algorithm that sorts the items in decreasing order of value per unit of weight ( $\delta = v/w$ ) and add an item to the knapsack at a time until resources are exhausted or all items are inserted [17]. We name the algorithm Path Adding Microservice Placement (PAMP) and its pseudocode is shown in Algorithm 2.

In what follows, we evaluate the effectiveness of PAMP algorithm with respect to complexity reduction and achievements of near-optimal placements.

##### C. COMPLEXITY ASSESSMENT

Computing the optimal value of  $S$  with an exhaustive search requires exploring  $2^{M-1}$  states, so the complexity grows exponentially as the size of the application increases. The complexity of the PAMP algorithm is a function of the number of dependency paths  $||P_i||$ . For highly meshed  $G_m$  dependency graphs,  $||P_i||$  can be close to  $2^{M-1}$ .<sup>11</sup> Practically, typical microservice applications do not have these extremely meshed dependency graphs, and the PAMP algorithm greatly reduces computational complexity.

To support this hypothesis, we consider the dependency graphs  $G_m$  of 30 microservice applications provided by the  $\mu$ Bench tool [4], which are derived from real Alibaba cloud traces [18], [19]. For each microservice application, the

<sup>10</sup>In a DAG, the search for all possible paths can be done in linear time by topological sorting and then using dynamic programming.

<sup>11</sup>E.g., consider a DAG for which node  $i$  has a link to node  $j$  if  $i < j$ . The number of dependency paths is  $2^{M-1} - 1$ .

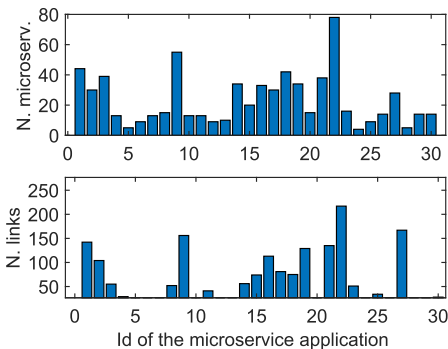


**Algorithm 2** Path Adding Microservice Placement (PAMP)

```

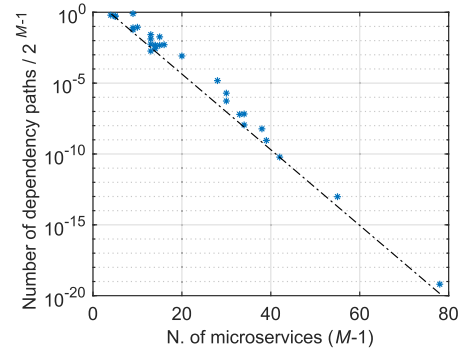
1: Compute the dependency paths  $\Pi$ 
2:  $\Pi_r = \Pi$   $\triangleright$  remaining paths to add
3:  $S_{opt}(\hat{U}) = 1$  for  $d_i = \text{cloud}$   $\triangleright$  initialize  $S_{opt}$ 
4:  $S_{opt}(\hat{U}) = 0$  for  $d_i = \text{edge}$ 
5:  $S_{opt}(\hat{U}) = 1$ 
6: while  $\Pi_r$  not empty do
7:   for path  $\pi \in \Pi_r$  do
8:      $S = S_{opt}$ 
9:      $S((i, \text{edge})) = 1, \forall i \in \pi$   $\triangleright$  New state with  $\pi$ 
10:    if  $S$  doesn't respect Eqs. 17,18,19 then
11:       $\delta(\pi) = -\infty$ 
12:    else
13:       $v(\pi) = 1/D_m(\hat{U})$   $\triangleright$  value
14:       $w(\pi) = R_{cpu}^{tot}/C_e + R_{mem}^{tot}/M_e$   $\triangleright$  weight
15:       $\delta(\pi) = v(\pi)/w(\pi)$   $\triangleright$  value per weight unit
16:    end if
17:  end for
18:  set  $\pi_{opt}$  as the path  $\pi$  with maximum  $\delta$ 
19:  if  $\delta(\pi_{opt}) == -\infty$  then
20:    break  $\triangleright$  edge resource exhaustion
21:  else
22:     $S_{opt}((i, \text{edge})) = 1, \forall i \in \pi_{opt}$   $\triangleright$  New opt state
23:    remove  $\pi_{opt}$  from  $\Pi_r$ 
24:  end if
25: end while
26: return  $S_{opt}$ 

```



**FIGURE 5.** N. of microservice and links of the microservice dependency paths  $G_m$  for the Alibaba-derived microservice applications offered by  $\mu$ Bench repository [4].

$\mu$ Bench repository contains some traces of calls made among microservices to serve user requests. Using these traces, we created the microservice dependency graph  $G_m$  of the application, inserting a link between the microservice  $i$  and  $j$  if there is at least one call from  $i$  to  $j$  in the traces. Fig. 5 shows the number of microservices and the number of links of the resulting dependency graph  $G_m$  of each application. We note a rather heterogeneous set of applications, ranging from a small application with 5 microservices and 7 links, to a medium-sized application with 78 microservices and 217 links.



**FIGURE 6.** Ratio between the dependency paths and the total number of possible placements representing a measure of the complexity reduction provided by PAMP algorithm with respect to the exhaustive search of the optimal placement for microservice applications derived by Alibaba traces in [4].

**TABLE 2.** Coconfigurations (cfg.) of Albert-Barabasi dependency graphs.

Cfg.	$\alpha$	$a$	$e$	
A	0.05	0.01	1	highly-centralized hierarchical applications
B	0.9	0.01	1	orchestrator-based applications
C	0.05	3.25	1	applications with several auxiliary services
D	0.9	3.25	1	multi-tier applications

For each application, we computed the ratio between the number of dependency paths and  $2^{M-1}$  as a measure of the reduction in complexity provided by PAMP. Fig. 6 shows that this ratio tends to decrease exponentially with a rate on the order of  $10^{-M/4}$ . Consequently, PAMP strongly reduces the computational complexity compared to exhaustive search, and this reduction is greatest when it is most needed, i.e., for applications with a large number of microservices.

**D. PERFORMANCE ASSESSMENT**

Besides complexity reduction, we measured the effectiveness of the PAMP algorithm by comparing its performance with that achievable with an exhaustive search of the optimal placement. For this evaluation, we generated the dependency graphs of microservice applications according to the Barabási-Albert (B-A) model [20]. The graph is built by adding one node at a time and connecting it to a single parent ( $e = 1$ ). A node  $i$  is chosen as the parent with a probability (unnormalized) equal to  $p = f_i^\alpha + a$ , where  $f_i$  is the number of nodes that have already chosen node  $i$  as the parent,  $\alpha$  is the power of preferential attachment and  $a$  is the attractiveness of nodes without children. Literature studies report that the B-A model fits well with the dependency graph of different classes of microservice applications [21]. Table 2 shows B-A parameters of 4 representative configurations named A, B, C and D. Fig. 7 shows an example of dependency graphs built using these parameters. We note that increasing the parameter  $\alpha$  increases the presence of few “hubs,” i.e., nodes with many children, because of the preferential attachment effect.

Increasing  $a$  mitigates this effect by making the graph more homogeneous and reducing the size of the hubs.

After generating the dependency graph with the B-A algorithm, we configured the values  $P_c^m(i, j)$  for all microservices  $i, j$  that have a dependency equal to a constant  $p$ , which we adjusted using Eq. 4 so that the average number  $m_{xc}$  of microservices involved per request is equal to 9. This value is approximately the median value of the measurements reported in [19].

Fig. 8 shows the average computation time of a placement solution as the number of microservices varies. Each value is an average computed over 20 applications, whose dependency graphs were generated with the B-A algorithm and with a configuration randomly chosen from those in Table 2. The absolute times depend on the power of the machine on which the computation is performed; however, as a general result, we note that the complexity of the exhaustive search grows exponentially, while that of the PAMP algorithm grows linearly. In fact, for dependency graphs generated with the B-A algorithm with only one parent per node ( $e = 1$ ), the number of dependency paths is equal to the number of microservices.

Fig. 9 shows the ratio between the delay of user requests using the PAMP placement algorithm and the minimum delay obtained from an exhaustive search for the optimal solution. As expected, PAMP is a suboptimal algorithm, since this ratio is greater than one. However, the relative increase in the request delay is limited to a few percentage points. This makes us confident that we can use the PAMP algorithm in the next section to draw reliable conclusions on the impact of properties of microservice applications on edge computing exploitability for reducing user delay, without the risk of possible bias due to placement algorithm inefficiencies.

## V. ANALYSIS OF RISKS, OPPORTUNITIES AND ENABLERS

In this section, we analyze the performance of microservice applications by varying the amount of resources available in the edge data center. For any considered application, running microservice instances at the edge exhausts CPUs before memory. Therefore, we consider the number of CPUs ( $C_e$ ) as a representative parameter of edge resources, as it is the only factor that limits edge deployment. When not explicitly stated, the parameter configurations are the default ones shown in Table 1.

The generic methodology used to perform the various analyzes is to consider a representative microservice application whose dependency graph ( $G_m$ ) is randomly generated through the B-A algorithm, use a placement strategy to select microservice instances to run in the edge, and then the model in Sec. II to evaluate performance. Reported results are the average of 20 trials and concern the following performance metrics:

- ratio between the user delay achieved with and without edge computing, where without edge computing means that microservices instances run only in the cloud;

- edge distribution factor  $\eta_e$ , defined as the ratio of the number of microservices running in the edge to the total number  $M - 1$  of microservices;
- edge the utilization factor  $\rho_e$  equal to the ratio between the number of CPUs used by edge microservice instances and the total number  $C_e$  of available CPUs;
- cloud-edge network traffic;
- average number of edge/cloud interactions  $N_{RT}$ .

## A. RISKS AND OPPORTUNITIES OF USING EDGE COMPUTING FOR MICROSERVICE APPLICATIONS

The first analysis compares the delay reduction provided by optimal placement strategy versus that provided by a greedy *random* strategy that executes in the edge a random subset of microservices that can saturate the edge's resources.<sup>12</sup>

The objective of the analysis is to show and comment on a rather surprising result, never presented in the literature before to the best of our knowledge, that is: with a well-designed placement strategy, edge computing offers an opportunity to reduce user delay, but with a wrong placement strategy, the use of edge computing risks to worsen delay performance compared to the case in which the application is executed only in the cloud.

To support this finding, we considered a microservice application consisting of 13 microservices. Each microservice requires a random amount of computing resources (CPUs), the total number of required CPUs is 13, and Fig. 10 shows the dependency graph among the microservices. Microservice 1 receives user requests, which arrive according to a Poisson process with an average arrival rate of 40 requests per second. Each microservice performs its internal function and then calls its downstream microservices sequentially, according to a call probability  $P_c^m$  equal to 0.79 so that the average number of microservices per call ( $m_{xc}$ ) is equal to 9.

The metric used to evaluate the effectiveness of a placement strategy is the ratio of user delay achieved with and without edge computing. Without edge computing, the placement strategy is not utilized, as all microservices are executed only in the cloud. Therefore, this metric measures the ability of the placement strategy to leverage edge computing to reduce delay. Note that if this ratio is less than one, the placement strategy reduces user delay; otherwise, it worsens it, i.e., with such a strategy it would be better not to use edge computing. Fig. 10b shows this delay ratio as edge computing resources (CPUs) increase.

We note that with a good placement strategy, such as optimal one,<sup>13</sup> the delay ratio is less than one, and thus the exploitation of edge resources results in a useful delay

<sup>12</sup>Specifically, the random algorithm repeatedly selects a microservice at random from those not yet running in the edge and for which the edge has enough resources. The loop ends when no other microservice can be selected due to the exhaustion of edge resources.

<sup>13</sup>We also considered the PAMP algorithm for placement, but since in this case the results are the same as the optimal ones, we do not show them in the plots.

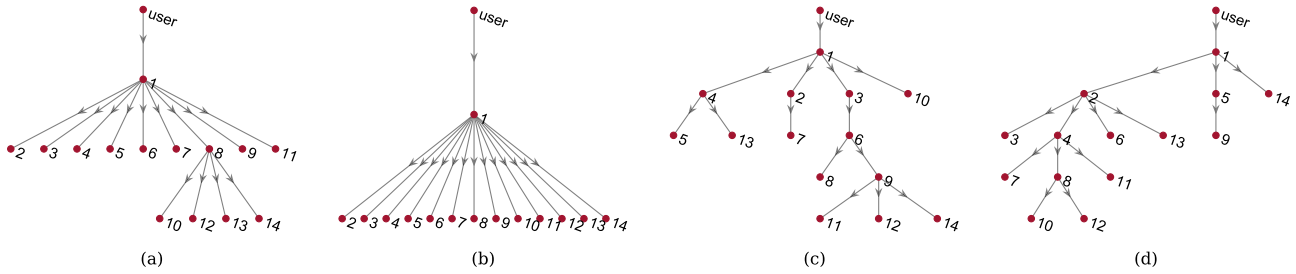


FIGURE 7. Examples of microservice dependency graphs  $G_m$  for configurations in Table 2.

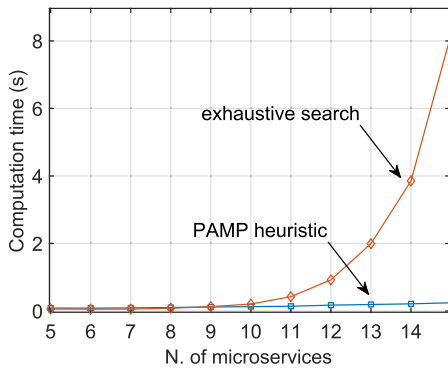


FIGURE 8. Time for computing placement solutions with 2,3 GHz 8-Core Intel Core i9 in case of PAMP algorithm and exhaustive search of the optimal solution showing the reduction of PAM computation time as a consequence of its reduced complexity.

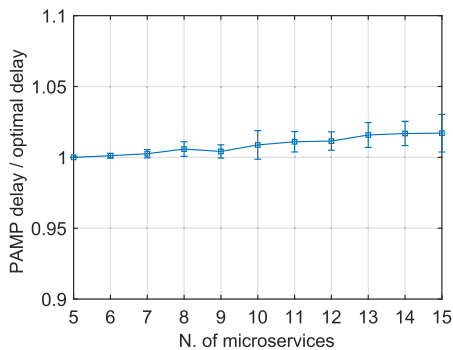


FIGURE 9. Average value and 95% error bars of the ratio between the user delay provided by placement solutions of PAMP algorithm and the minimum user delay provided by the optimal placement. Results show the near-optimal behavior of the PAMP algorithm in minimizing the user delay.

reduction. In contrast, a bad placement strategy, such as the random one, risks making the delay greater than if no edge computing is used; and the worsening of performance occurs in the middle of Fig. 10b, because in the extreme configurations of 0 or 13 CPUs, none or all of the microservices can be executed on the edge, and therefore the performance of the random and optimal strategy is the same.

The reason why a not-well-designed placement strategy can so critically affect user delay is the fact that a bad choice of microservice instances to run in the edge can create the

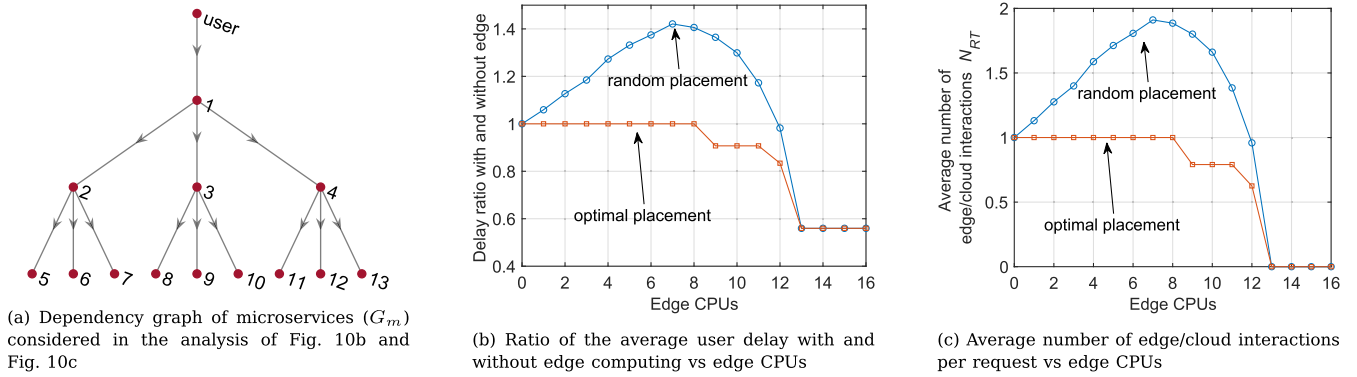
need to perform many interactions between microservices in the edge and cloud to resolve a request. This occurrence drastically worsens the delay performance compared to a cloud-only execution of the application, for which the number of edge/cloud interactions is always equal to 1.<sup>14</sup> For example, consider the case of a placement solution for the application in Fig. 10a for which microservice 1 runs in the edge while all others run only in the cloud. To serve a user request, the edge instance of microservice 1 sequentially calls the cloud instances of microservices 2, 3, and 4. This results in 3 edge/cloud interactions to resolve the request. Conversely, if the entire application ran only in the cloud, there would be only a single edge/cloud interaction between the user and the microservice 1. Therefore, the placement of only microservice 1 in the edge is a bad choice that worsens user delay as it increases the number of edge/cloud interactions to solve a request. Our hypothesis is confirmed by the results shown in Fig. 10c, in which we see that a not-well-designed placement strategy such as the random one creates an average number of edge/cloud interactions greater than one, which worsens the delay performance with respect to a cloud-only execution of the application (Fig. 10b).

In contrast, when there are sufficient resources (i.e., edge CPUs > 8), a well-designed strategy such as the optimal strategy (or even PAMP) places in the edge a group of microservices that are able to locally resolve a quota of service requests while avoiding contacting instances of microservices running in the cloud. As a result, the average number of edge/cloud interactions becomes less than one (Fig. 10c) and thus the user delay is less than that provided by a cloud-only execution of the application (Fig. 10b). As the number of edge CPUs increases, the strategy may have the opportunity to execute additional groups of microservices able to locally process user requests, and thus the quota of locally resolved requests increases, thereby further reducing user delay.

The results of this analysis showed the following.

- The design of placement strategies for microservices is extremely critical. In fact, by using a well-designed placement strategy, edge computing is a friend of microservice applications, i.e., it provides the opportunity to reduce user delay by leveraging edge resources

<sup>14</sup>We recall that we are assuming that user requests are always routed through the edge data center.



**FIGURE 10. Comparison of performance achieved with optimal and random placement strategy versus edge CPUs.**

compared to not using edge resources. In contrast, using strategies that are not-well-designed may instead turn edge computing into a foe of microservice applications, i.e., exploitation of edge resources with such strategies risks to increase user delays compared to the case of not using edge resources.

- Microservice developers should try to “cluster” the application into possibly small groups of microservices that can independently resolve a request, so that it is convenient and feasible to activate them in the edge, even in the case of limited resources. In a way, we can identify this microservices clustering approach as an edge-native design style, in that it enables a microservice application, combined with a well-designed placement strategy, to effectively leverage edge computing to reduce user delay.

**B. APPLICATION-LEVEL EDGE COMPUTING ENABLING FACTORS**

In this section, we analyze application performance using the PAMP placement strategy and varying an application-level property at a time. The properties examined are the number of microservices, the topology of the microservice dependency graph, the number of microservices involved per request, and the presence of databases and data caches. The analysis reveals which configuration of these properties, combined with an optimal or near-optimal strategy, enables a more significant reduction in user delay by using edge resources. The findings of the analysis can be used in the design phase of microservice applications to make them more edge-native, that is, providing placement strategies with the possibility of more significantly reducing user delay by using edge resources.

**1) NUMBER OF MICROSERVICES**

In this paragraph, we analyze whether the decomposition of the workload of an application into a set of microservices is a convenient design style to allow the reduction of user delay by exploiting edge resources.

To perform the analysis, we considered applications with different numbers of microservices, 10, 30, and 50, and to make a fair comparison, we scaled the number of CPUs required by microservices ( $R_{cpu}$ ) so that the compared applications required, on average, the same total amount of CPUs equal to 31.

Fig. 11 and Fig. 12 show the result of this analysis. Each figure presents the values of a different performance metric with respect to the number of edge CPUs and for microservice applications with a different number of microservices ( $M - 1$ ).

First, we discuss the impact of edge CPUs on performance. Fig. 11a shows that the increase in edge CPUs allows the placement strategy to increase the number of microservices executed in the edge data center, that is, an increase in the distribution factor ( $\eta_c$ ), which grows slowly at first, then, in the middle area of the plot, the increase is more than linear to saturate at the end when all microservices are executed in the edge.

Fig. 11b shows the increase in edge utilization factor  $\rho_e$  with edge CPUs. We recall that  $\rho_e$  is the total CPU consumed by edge microservices “normalized” to the number of edge CPUs. Therefore, its growth with edge CPUs means that the number of microservices instantiated to the edge increases more than linearly, as also confirmed by  $\eta_c$  (Fig. 11a). As a result, the available edge resources are used more efficiently when there are many of them. Obviously, when all microservices are executed on the edge, additional edge CPUs are unused, and thus the utilization factor starts to decrease.

Fig. 11c shows the ratio between the average delays with and without edge computing. As edge CPUs increase, more and more microservices are activated in the edge by the placement strategy, and this reduces the delay compared to the case of a cloud-only execution of the application. When all microservices are executed in the edge, the curves flatten because the delay reaches the lower bound equal to the sum of the execution times ( $D_i$ ) of the internal functions of all microservices involved in a request. Fig. 12 shows a similar behavior for cloud-edge network traffic ( $T_{nce}$ ), which drops to zero when the entire application is executed in the edge.

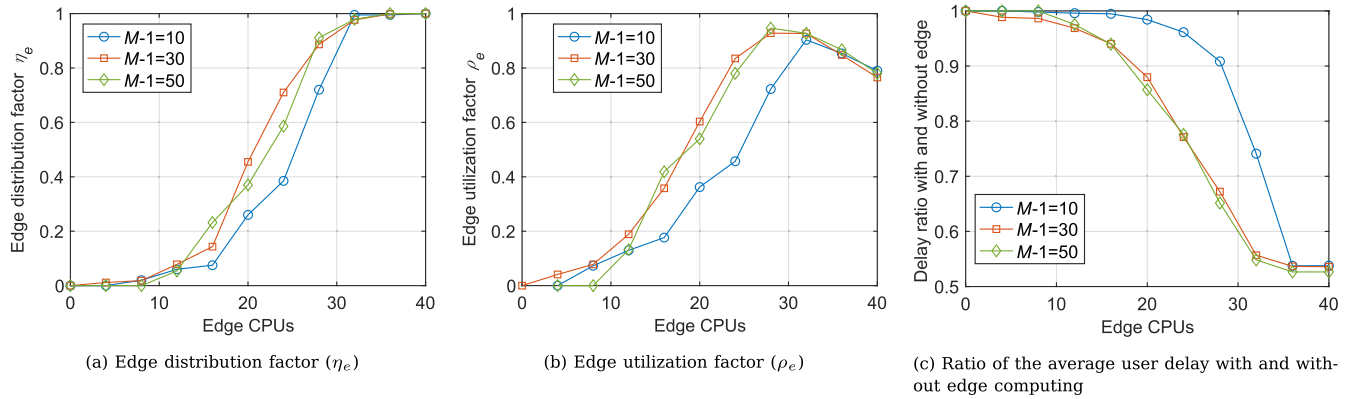


FIGURE 11. Edge computing performance for applications with different numbers of microservices ( $M - 1$ ).

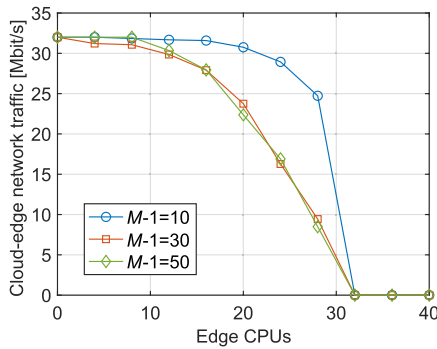


FIGURE 12. Cloud-edge network traffic for applications with different numbers of microservices ( $M - 1$ ).

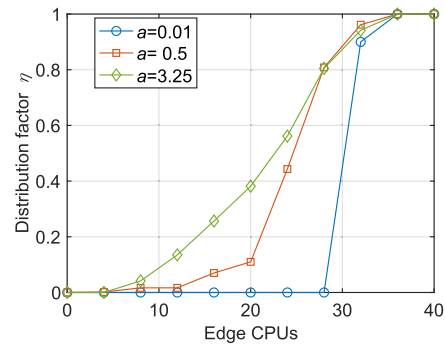


FIGURE 13. Edge distribution factor for applications whose dependency graphs  $G_m$  differs in the B-A  $\alpha$  parameter.

Let us now discuss the impact of the number of microservices on performance. If we had a monolithic application, i.e., a single microservice requiring 31 CPUs, we would be able to instantiate that giant microservice in the edge when the edge CPUs are greater than or equal to 31. Therefore, for smaller values of edge CPU values, edge computing can not be used. In contrast, the splitting of the workload of a monolithic application into microservices, i.e., an increase in the number of microservices, allows only a portion of the application to be executed in the edge, thus enabling the exploitation of edge computing resources for reducing user delay even in resource-limited settings.

This insight is confirmed by the behavior of  $\rho_e$  in Fig. 11b, delay ratio in Fig. 11c and network traffic in Fig. 12. Applications with 10 microservices have a lower  $\rho_e$  value than applications with more microservices. This is due to the fact that placement strategies have more difficulty in activating the microservices of applications with 10 microservices in the edge data center because, being fewer in number than those of the other applications considered, each of them requires more CPU and is more difficult to fit into the edge resources. This lower ability to exploit edge resources leads applications with 10 microservices to have worse delay and traffic performance, as shown in Fig. 11c and Fig. 11c, respectively.

Applications with 30 or 50 microservices behave similarly because in both cases the microservices are tiny enough to be smoothly fit into the available edge resources.

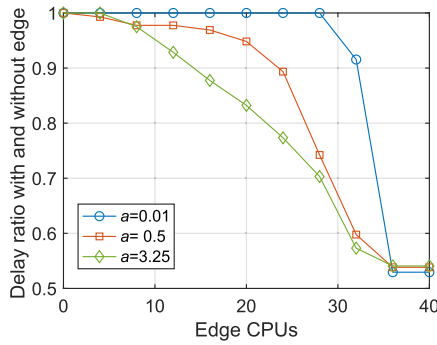
The result of this analysis showed that the decomposition of the workload of an application into microservices improves the possibility of using edge computing resources to reduce user delay.<sup>15</sup> In this sense, the architectural style of structuring an application into microservices is an enabling factor of edge computing.

## 2) TOPOLOGY OF THE MICROSERVICE DEPENDENCY GRAPH

In this section, we analyze how the characteristics of the dependency graph of an application impact the possibility of a placement strategy to take advantage of edge resources to reduce delays.

To carry out the analysis, we considered applications made by 30 microservices and three classes of dependency graphs, which follow the B-A model presented in Sec. IV and differ in the attractiveness of nodes without children ( $a$ ); the other parameter  $\alpha$  is set equal to 0.9. The first class has  $a = 0.01$

<sup>15</sup>We talk about “possibility” because the delay reduction actually takes place only using a well-designed placement strategy, as in the case of our PAMP strategy.



**FIGURE 14.** Ratio of the average user delay with and without edge computing for applications whose dependency graphs  $G_m$  differs in the B-A  $\alpha$  parameter.

and is representative of applications with highly centralized dependency graphs, such as those in Fig. 7a and Fig. 7b. The last class has  $a = 3.25$  and consists of applications with highly distributed dependency graphs, such as those in Fig. 7c and Fig. 7d. The second class consists of applications whose topology has an intermediate level of distribution.

Fig. 13 and Fig. 14 show the edge distribution factor and the delay ratio with and without edge computing of the three application classes, respectively. The results in Fig. 13 reveal that a placement strategy has more difficulty in executing in the edge applications made of few hub microservices with a very large number of downstream microservices, such as those with  $a = 0.01$ , than applications with more distributed dependency graphs, such as those with  $a = \{0.5, 3.25\}$ , in which the microservices have a smaller fanout. And the lower edge exploitation also implies a lower delay reduction with respect to the case of cloud-only execution, as shown in Fig. 14.

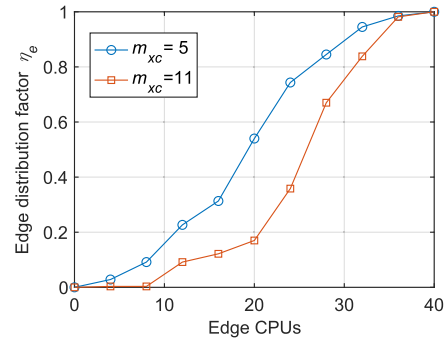
This can be explained by the following example. If the placement strategy executes in the edge a microservice  $i$  that has a number  $F$  of downstream microservices surely called to serve a request, this microservice would create  $F$  edge-cloud interactions per request, thus worsening the delay. It is convenient to execute the microservice  $i$  in the edge only when all its downstream microservices can also be activated in the edge. This requirement makes it more difficult to use edge resources for applications that have microservices with high fanout.

Consequently, this analysis showed that designing microservice applications with a more distributed dependency graph with low-fanout microservices is an enabling factor of edge computing, in the sense that increases the possibility of reducing user delay by exploiting edge resources.

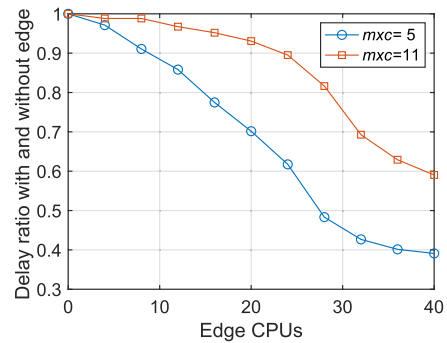
### 3) NUMBER OF INVOLVED MICROSERVICES PER REQUEST

In this analysis, we analyze the performance of microservice applications made of 30 microservices that involve a different average number of microservices per request ( $m_{xc}$ ).

Fig. 15 shows the edge distribution factor versus  $m_{xc}$ . The results show that microservice applications that use more



**FIGURE 15.** Edge distribution factor for applications with different number of microservices per request  $m_{xc}$ .



**FIGURE 16.** Ratio of the average user delay with and without edge computing for applications with different numbers of microservices per request  $m_{xc}$ .

microservices per request ( $m_{xc}$ ) exploit edge resources worse, i.e., the placement strategy executes a lower quota of them in the edge data center. Consequently, the delay reduction with respect to the case of not using edge computing is also worst, as shown in Fig. 16.

We motivate this behavior as follows. As discussed earlier, to take advantage of edge computing, microservices usually need to be instantiated in groups that are rather autonomous in serving a request to avoid interactions with the cloud. At higher  $m_{xc}$ , the microservices interoperate more with each other; the autonomous groups are larger, making it more difficult to activate them with limited edge resources.

Consequently, this analysis showed that designing an application with a small number of microservices involved per request is an enabling factor to allow the reduction of user delay by exploiting edge computing resources.

### 4) PRESENCE OF DATABASES

Microservice applications often have a centralized database that microservices use to store and read global data [22], [23], [24]. This database can only run in the cloud because its replication in edge data centers could create data consistency problems. In this paragraph, we analyzed whether this database has an impact on the possibility of a placement strategy to use edge resources for reducing user delay.

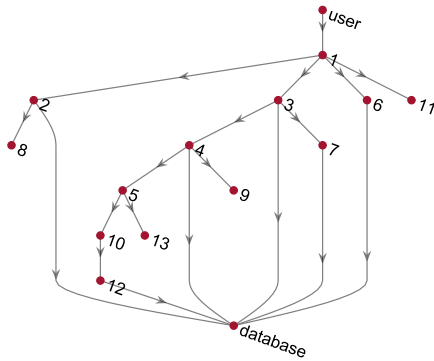


FIGURE 17. Dependency graph of microservices ( $G_m$ ) with database.

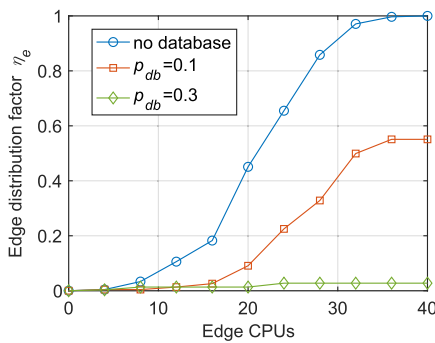


FIGURE 18. Edge distribution factor for applications without cloud database and with cloud database for different percentage  $p_{db}$  of microservices using it.

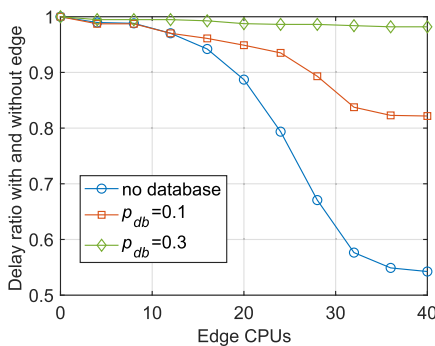


FIGURE 19. Ratio of the average user delay with and without edge computing for applications without cloud database and with cloud database for different percentage  $p_{db}$  of microservices using it.

To undertake the analysis, we modeled the constraint of running the database only in the cloud with Eq. 20. We modified the B-A algorithm to build the dependency graph so that the last microservice is the database and the other microservices have the database among their downstream microservices with a probability equal to  $p_{db}$ . Furthermore, if a microservice  $i$  has the database among its downstream microservices, the relative call probability  $P_c^m(i, \text{database})$  is equal to 1, simulating the fact that the microservice  $i$  always needs the database data to process a request. Fig. 17 depicts

an example of a possible dependency graph in the presence of a database.

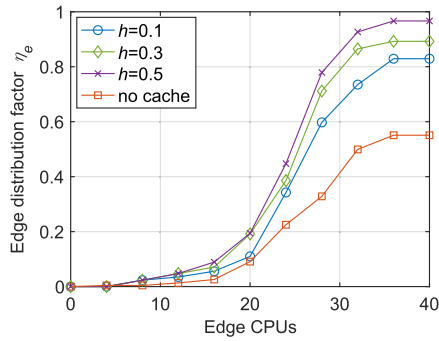
Fig. 18 shows the edge distribution factor for applications made up of 30 microservices (including the database), in the absence and presence of a database for  $p_{db} = \{0.1, 0.3\}$ . The results reveal that the presence of a centralized database severely reduces the possibility of using edge resources to reduce user delay, even for a small percentage  $p_{db}$  of microservices that use it. For example, when we have 40 CPUs, without a database, the placement strategy runs the entire application in the edge ( $\eta_e = 1$ ). On the contrary, when 10% of the microservices use the database ( $p_{db} = 0.1$ ), the placement strategy runs only 55% of the application in the edge and almost nothing when  $p_{db}$  grows further. As a consequence of fewer microservices running in the edge, the presence of the database dramatically worsens the delay reduction achieved by using edge computing compared with that achieved by not using edge resources, as shown in Fig. 19.

The reason for this dramatic impairment provided by centralized databases is due to a kind of “gravitational” effect that the database creates around it, which attracts microservices that use the database, as well as their children and parents, to stay close to it in the cloud. This effect can be motivated as follows. It is useless to run an instance of microservice  $i$  that uses the database in the edge because this action does not save any edge-cloud round trip. This effect also propagates around  $i$ , in the upstream and downstream direction of the dependency graph. Since  $i$  remains in the cloud, according to the necessary condition 1, all of its children which have  $i$  as their only parent must also remain in the cloud. Moreover, the parents of  $i$  that call it with high probability  $P_m^c$  remain in the cloud as well because their execution on the edge would create a non-beneficial interaction between edge and cloud. This reasoning can be repeated recursively for children of children and parents of parents, thus expanding the gravitational effect of the database both upstream and downstream of  $i$  in the dependency graph. This gravitational effect of the database is very strong and grows as the number of microservices that use the database increases.

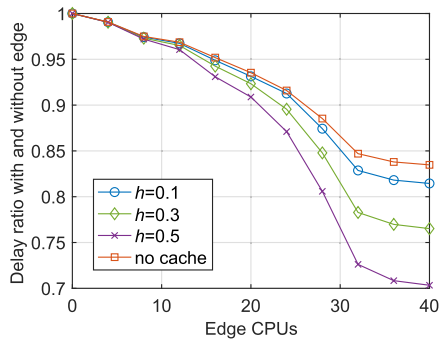
Consequently, the results of this analysis showed that the design of microservice applications with centralized databases in a *not* enabling factor for the exploitability of edge computing, i.e., the presence of a centralized database seriously jeopardizes the possibility of using edge resources to reduce user delay.

### 5) DATA CACHING

We note that many microservice applications use caching systems [25], such as Memcached or Redis, to limit database access and speed up data retrieval. We argue that these caching systems can also effectively mitigate the negative effect that a centralized database brings in the exploitability of edge resources for delay reduction. Indeed, data caching can avoid going to the cloud every time whenever there



**FIGURE 20.** Edge distribution factor for applications with database ( $\rho_{db} = 0.1$ ) and data caches for different cache hit probabilities ( $h$ ).



**FIGURE 21.** Ratio of the average user delay with and without edge computing for applications with database ( $\rho_{db} = 0.1$ ) and data caches for different cache hit probabilities ( $h$ ).

is a need for database data. Accordingly, in this section, we support this hypothesis by analyzing the performance of microservice applications made of 30 microservices that use a centralized database and also data caching. Performance is measured by varying the effectiveness of the cache, that is, the probability  $h$  of finding requested data in a local cache that each microservice is assumed to have, also known as the cache hit probability [26].

To carry out the analysis, we modeled the presence of a local cache as follows. A microservice  $i$  that has the database as a downstream microservice calls it with probability  $P_c^m(i, \text{database}) = 1 - h$

Fig. 20 shows the edge distribution factor when 10% of the microservices use the database and in the presence of local microservice caches with different cache hit probabilities  $h$ . We note that an increase in the cache hit probability leads to an increase in the number of microservices that the placement strategy executes in the edge. Consequently, as shown by Fig. 21 there is a better reduction in user delay. However, even in the best case of 40 CPUs and  $h = 0.5$ , although Fig. 20 shows that all microservices, except for the database, are executed in the edge, the delay ratio is on the order of 70%, compared to the better 55% in the absence of database (Fig. 19). In fact, a quota of requests must still reach the database in the cloud being  $h < 1$  thus increasing the average user delay.

Overall, this analysis has shown that the introduction of local caching systems can mitigate the difficulties of using resources at the edge to reduce delays for microservice applications that use centralized databases. However, even for a very ambitious cache hit probability of 0.5, the penalty of having a central database can not be completely eliminated.

## VI. RELATED WORKS

To the best of our knowledge, this is the first work that provides insight to microservice developers on application-level parameters that influence the placement of microservices at the edge and thus the ability to use edge resources to reduce user delay. To achieve this contribution, we had to develop a new delay model and a PAMP placement strategy because those we found in the literature did not take into account key aspects of the targeted analysis all together, such as the presence of a cloud back-end where microservice instances are always executed, routing policies of the service mesh, the random spanning of the dependency graph to simulate the involvement of different microservices per call, sharing of CPU and network resources among multiple requests and their modeling through queuing systems with processor sharing, and the presence of databases and data caches. In the next subsections, we motivate these statements by revising related works and discussing the differences with respect to our contributions.

### A. MICROSERVICE PLACEMENT STRATEGIES AND DELAY MODELS

Microservice placement is a research topic of great interest in the literature, as it involves finding the optimal location for each microservice to run efficiently and effectively. Placement strategies can take advantage of both cloud and edge infrastructures. Cloud infrastructures provide the flexibility and scalability needed to handle large workloads, while edge infrastructures bring services closer to end users, potentially reducing latency and improving response times [8], [9]. Many studies propose methods to control resources to avoid QoS violations and take full advantage of cloud services [27], [28], [29], [30], [31], [32], [33]. When the scenario is extended by adding peripheral data centers, the methods become more complex due to the possible heterogeneity of resources, the presence of a network between data centers, etc.

In [34], the authors propose an online placement algorithm for the deployment of applications consisting of service graphs in a hierarchical cloud scenario. While our objective function is delay minimization, their goal is load balancing among the nodes of the cloud hierarchy. Also, probabilistic dependency graphs are not considered, i.e., the fact that the microservices involved for each request may be randomly different. It is also assumed that a microservice is executed in a single node, whereas we consider the possibility of running a microservice in both edge and cloud, and this also led us to consider the service mesh routing policy.



In [35], the authors propose a Bayesian Optimization-based iterative reinforcement learning algorithm for the placement of containerized microservices in IoT considering the time-varying resource availability. Their objective function is to minimize the delay of the worst microservice, rather than the delay experienced by the user as in our case. Moreover, they assume that a microservice can run only in a single (fog) node.

In [36], the authors propose a learning-based mechanism to proactively place microservices on edge servers considering a “linear” microservice dependency graph and leaving for future work the case of DAG dependency graphs as in our case. The authors formulated the problem as a Markov decision process in which at the beginning of each time slot the agent observes the usage of microservice applications and can perform one of the following actions: proactively place all microservices of the chain in neighboring edge servers, migrate microservices from one edge server to another because the requesting user has changed location. The problem is quite different in that we are not considering user mobility, and we are not limiting the placement options to only two options, but we are considering the best placement.

In [37], the authors jointly consider the problem of placement and on-demand scheduling of dependent tasks. If a task is not present on edge nodes, it is transferred from the cloud and executed at the edge, in case replacing unneeded tasks. An edge task is used by only one request, unlike the case of a microservice instance that is used by many requests concurrently. As a result, the authors do not need to account for processor-sharing issues of computing resources, and the internal processing of a microservice lasts a constant time independent of load. In addition, the authors do not model the random spanning of the dependency graph and the sharing of network resources. However, unlike us, their work addressed a multi-edge placement problem although, in our opinion, more task-oriented than microservice-oriented.

In [38], the authors consider a multi-edge scenario with a cloud back-end running microservice applications. Their microservice application model is quite different from ours. Based on experimental results [39] and demo microservice applications [4], [22], [40], [41], [42], we modeled microservice interactions with a randomly spanned DAG whose interactions between parent and child nodes occur sequentially and follow a request-response pattern, for which the child always responds to the parent and the parent responds to its parent when all its children have responded. The presence of request-response interactions (as in the typical case of REST APIs) and the DAG form of the call graph create many edge-cloud round trips if the placement strategy is not optimized accordingly. The application model considered in [38], on the other hand, is very different and much representative of “task pipelines” in which there is a sequence of tasks to be performed and the output of one task is the input of the next task. There is never any back and forth between child and parent, only moving forward. To account for the fact that the user

must receive feedback, the output of the last task is sent to the user. Also in this paper, processor-sharing is not considered since it is assumed that a task/microservice serves one request at a time, unlike our microservice work model, in which it concurrently serves multiple requests. Also, the sharing of network resources, and thus the impact of network load on delay, is not considered.

## B. PLACEMENT STRATEGIES AND DELAY MODELS FOR SERVICE FUNCTION CHAINING

Studies in the literature that are close to the world of microservices are those that concern the area of service function chaining (SFC), where services are virtual network functions (VNFs) that process “traffic flows” through them [43]. The placement problem is to determine the best set of network nodes to execute the VNFs required by a flow, considering both the availability of computing and network resources and the optimizations of network routing [44]. However, the service model is different from that of microservice applications. There is no request-response interaction between successive VNFs as there is between microservices. There are no dependency graphs but simpler dependency chains. Usually, a VNF serves one flow, whereas a microservice instance serves multiple requests at a time with a processor-sharing approach. Flows can be served with request quality or not served at all in the absence of the required resources for their VNFs, differently, microservice requests are always served and at most slowed down in the presence of limited resources to share like CPU quota or network connection.

## C. DATA CHAINING FOR MICROSERVICE APPLICATIONS

Other works related to this paper are those addressing solutions for data caching in the case of microservice and/or edge computing applications. For example, in [45], the authors advocate the importance of data caching in microservice applications and propose a new replacement policy, whose performance has been evaluated through a simulation assuming that the application runs in a single data center and taking into account the microservice dependency graph. In [46], the authors extend the caching scenario to a set of distributed edge servers and address the Edge Data Integrity (EDI) problem. Analyzing the threat model and audit objectives, they propose a probabilistic lightweight sampling-based approach to verify data integrity in distributed edge caches. In [47], the authors address the problem of cooperative caching among a set of edge nodes and propose an online algorithm based on Lyapunov optimization to minimize data latency. In [25], the authors discuss the pros and cons of introducing caching in microservice applications, where the cons are mainly due to incomplete support of application protocols, such as gRPC, and data consistency issues. The authors develop and implement caching in service meshes [7], which also works with the gRPC protocol; the mechanism is application independent (based on sidecar patterns such as Istio [14]) and therefore does not require changes in the source code. The authors

measured a valuable reduction in network traffic with a real microservice application.

## VII. CONCLUSION

In this work, we analyze the properties of microservice applications that can affect the ability of a placement strategy to deploy microservices in edge data centers to reduce the average user delay. The analysis is supported by a new analytical model of delay and a new placement strategy, called PAMP, which takes into account key aspects of the application, and of the supporting computing and network systems not considered all together in previous literature models. Our findings are as follows.

- Decomposing an application's workload into microservices improves the ability to leverage edge computing for reducing user delay, as it allows only part of the application to run in the edge when the edge's resources are insufficient to run the entire application (Fig. 11c).
- The use of edge resources decreases user delays when it reduces the average number of network interactions between edge and cloud required per request (Fig. 10b, Fig. 10c). This observation leads to the conclusion that it is convenient to design applications consisting of *autonomous groups* of microservices that can independently resolve a quota of user requests, since these groups can be deployed in the edge when available resources permit and resolve user requests locally, reducing the need to use the network to reach the cloud.
- Applications whose microservices have a more distributed dependency graph or fewer microservices involved per call have a better possibility of using edge resources to reduce user delay as the autonomous microservice groups are likely smaller and thus easier to deploy in case of limited edge resources (Fig. 14, Fig. 16).
- The presence in the microservice architecture of a centralized database that must necessarily run in the cloud drastically reduces the possibility of gaining an advantage from edge computing for delay reduction (Fig. 19). In fact, it is not convenient to deploy in the edge those microservices (as well as all the related downstream microservices and part of their upstreams) that require database data to resolve a request, since this would create edge/cloud interactions that nullify the effectiveness of using edge resources to reduce user delay. Figuratively, the database has an effect similar to that of a gravitational mass that prevents microservices close to it from leaving the cloud, i.e., having user delay benefits by moving them to the edge data center.
- When it is not convenient to avoid a centralized database, for example, because it strongly simplifies implementation, it would be better to use local data caching systems that reduce the need for interaction with the database, thereby allowing the advantages of edge computing to reduce delay to be partially recovered (Fig. 21).

In addition to these application-level considerations, the analysis also highlighted some critical issues in the use of edge computing for microservice applications, which relate to the "mandatory" use of well-designed placement strategies. In fact, the use of optimal or near-optimal placement strategies provides the opportunity of using edge computing for reducing user delay. However, the use of a not-well-designed strategy that places the wrong set of microservices in the edge can even risk making worse delay performance than in the case of not using edge computing (Fig. 10b). In that sense, edge computing can be a friend or a foe of microservice applications, depending on the skill of the placement strategy.

Our work has a narrow focus, and therefore future work can extend it beyond the following limitations. We refer primarily to widespread microservice applications, used by a large set of fixed and mobile clients, for which the geographic pattern of requests is stable enough so that user mobility is not critical for the microservice placement. There are no edge-to-edge interactions, only edge-cloud interactions. We mainly considered applications based on request-response interactions rather than streaming; that is, microservice requests are of short duration.

## ACKNOWLEDGMENT

The authors would like to thank the stimulating conversations with Ludovico Funari and Luca Petrucci, who provided helpful comments for the research direction.

## REFERENCES

- [1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present And Ulterior Software Engineering*. Springer, 2017, pp. 195–216.
- [2] J. Thönes, "Microservices," *IEEE Softw.*, vol. 32, no. 1, p. 116, Jan. 2015.
- [3] C. Richardson, *Microservices Patterns: With Examples Java*. New York, NY, USA: Simon and Schuster, 2018.
- [4] A. Detti, L. Funari, and L. Petrucci, " $\mu$ Bench: An open-source factory of benchmark microservice applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 3, pp. 968–980, Mar. 2023.
- [5] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, 2014.
- [6] *Kubernetes: Production-Grade Container Orchestration*. Accessed: Mar. 21, 2023. [Online]. Available: <https://kubernetes.io/>
- [7] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *Proc. IEEE Int. Conf. Service-Oriented Syst. Eng. (SOSE)*, Apr. 2019, pp. 122–1225.
- [8] 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 Trans. Services Comput.*, vol. 15, no. 3, pp. 1746–1759, May 2022.
- [9] 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 Trans. Mobile Comput.*, vol. 20, no. 5, pp. 1907–1923, May 2021.
- [10] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2333–2345, Oct. 2018.
- [11] Y. Li, W. Dai, X. Gan, H. Jin, L. Fu, H. Ma, and X. Wang, "Cooperative service placement and scheduling in edge clouds: A deadline-driven approach," *IEEE Trans. Mobile Comput.*, vol. 21, no. 10, pp. 3519–3535, Oct. 2022.
- [12] S. Fan, I. Hou, V. S. Mai, and L. Benmohamed, "Online service caching and routing at the edge with unknown arrivals," in *Proc. IEEE Int. Conf. Commun.*, May 2022, pp. 383–388.

- [13] A. O. Duque, C. Klein, J. Feng, X. Cai, B. Skubic, and E. Elmroth, "A qualitative evaluation of service mesh-based traffic management for mobile edge cloud," in *Proc. 22nd IEEE Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, May 2022, pp. 210–219.
- [14] *Istio Service Mesh*. Accessed: Mar. 21, 2023. [Online]. Available: <https://istio.io/>
- [15] *Linkerd Service Mesh*. Accessed: Mar. 21, 2023. [Online]. Available: <https://linkerd.io/>
- [16] L. Kleinrock, *Queueing Systems: Computer Applications*, vol. 2. Hoboken, NJ, USA: Wiley, 1976.
- [17] J. Csirik, "Heuristics for the 0–1 min-knapsack problem," *Acta Cybernetica*, vol. 10, nos. 1–2, pp. 15–20, 1991.
- [18] *Cluster Trace Microservices v2021*. Accessed: Mar. 21, 2023. [Online]. Available: <https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021>
- [19] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2021, pp. 412–426.
- [20] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, Oct. 1999.
- [21] V. Podolskiy, M. Patrou, P. Patros, M. Gerndt, and K. B. Kent, "The weakest link: Revealing and modeling the architectural patterns of microservice applications," in *Proc. 30th Annu. Int. Conf. Comput. Sci. Softw. Eng. (CASCON)*. New York, NY, USA: ACM, 2020, pp. 113–122.
- [22] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A micro-service reference application for benchmarking, modeling and resource management research," in *Proc. IEEE 26th Int. Symp. Modeling, Anal., Simulation Comput. Telecommun. Syst. (MASCOTS)*, Sep. 2018, pp. 223–236.
- [23] Y. Gan, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 3–18.
- [24] R. Jung and M. Adolf, "The JPetStore suite: A concise experiment setup for research," in *Proc. Symp. Softw. Perform.*, 2018, pp. 1–3.
- [25] L. Larsson, W. Tärneberg, C. Klein, M. Kihl, and E. Elmroth, "Adaptive and application-agnostic caching in service meshes for resilient cloud applications," in *Proc. IEEE 7th Int. Conf. Netw. Softwarization (NetSoft)*, Jun. 2021, pp. 176–180.
- [26] N. B. Melazzi, G. Bianchi, A. Caponi, and A. Detti, "A general, tractable and accurate model for a cascade of LRU caches," *IEEE Commun. Lett.*, vol. 18, no. 5, pp. 877–880, May 2014.
- [27] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, Nov. 2020, pp. 805–825. [Online]. Available: <https://www.usenix.org/conference/osdi20/presentation/qiu>
- [28] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-based and QoS-aware resource management for cloud microservices," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 167–181, doi: [10.1145/3445814.3446693](https://doi.org/10.1145/3445814.3446693).
- [29] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 19–33, doi: [10.1145/3297858.3304004](https://doi.org/10.1145/3297858.3304004).
- [30] 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 Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2968–2978, Nov. 2022.
- [31] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and scalable ML-driven performance debugging in microservices," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 135–151, doi: [10.1145/3445814.3446700](https://doi.org/10.1145/3445814.3446700).
- [32] M. R. Hossen, M. A. Islam, and K. Ahmed, "Practical efficient microservice autoscaling with QoS assurance," in *Proc. 31st Int. Symp. High-Perform. Parallel Distrib. Comput.*, Jun. 2022, pp. 240–252.
- [33] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Trans. Cloud Comput.*, vol. 10, no. 2, pp. 1100–1116, Apr. 2022.
- [34] 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.
- [35] S. B. Nath, S. Chattopadhyay, R. Karmakar, S. K. Addya, S. Chakraborty, and S. K. Ghosh, "PTC: Pick-test-choose to place containerized microservices in IoT," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2019, pp. 1–6.
- [36] K. Ray, A. Banerjee, and N. C. Narendra, "Proactive microservice placement and migration for mobile edge computing," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, Nov. 2020, pp. 28–41.
- [37] 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 *Proc. Int. Symp. Quality Service*, Jun. 2019, pp. 1–10.
- [38] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar, "Distributed redundant placement for microservice-based applications at the edge," *IEEE Trans. Services Comput.*, vol. 15, no. 3, pp. 1732–1745, May 2022.
- [39] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, and C. Xu, "An in-depth study of microservice call graph and runtime performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3901–3914, Dec. 2022.
- [40] *DeathStarBench*. Accessed: Mar. 21, 2023. [Online]. Available: <https://github.com/delimitrou/DeathStarBench>
- [41] *Sock Shop: Microservices Demo*. Accessed: Mar. 21, 2023. [Online]. Available: <https://microservices-demo.github.io/>
- [42] *Distributed Version of the Spring Petclinic Adapted for Cloud Foundry and Kubernetes*. Accessed: Mar. 21, 2023. [Online]. Available: <https://github.com/spring-petclinic/spring-petclinic-cloud>
- [43] T. V. Doan, G. T. Nguyen, M. Reisslein, and F. H. P. Fitzek, "SAP: Subchain-aware NFV service placement in mobile edge cloud," *IEEE Trans. Netw. Service Manage.*, vol. 20, no. 1, pp. 319–341, Mar. 2023.
- [44] R. Chen and J. Zhao, "Scalable and flexible traffic steering for service function chains," *IEEE Trans. Netw. Service Manage.*, vol. 19, no. 3, pp. 2048–2062, Sep. 2022.
- [45] L. Li, C. Ye, and H. Zhou, "Cache replacement algorithm based on dynamic constraints in microservice platform," in *Proc. Int. Conf. Service Sci. (ICSS)*, May 2022, pp. 167–174.
- [46] B. Li, Q. He, F. Chen, H. Jin, Y. Xiang, and Y. Yang, "Auditing cache data integrity in the edge computing environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1210–1223, May 2021.
- [47] X. Xia, F. Chen, Q. He, J. Grundy, M. Abdelrazek, and H. Jin, "Online collaborative data caching in edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 281–294, Feb. 2021.



**ANDREA DETTI** (Member, IEEE) is currently a Professor in wireless networks and cloud computing with the Department of Electronic Engineering, University of Rome "Tor Vergata." He is the coauthor of many papers in journals and conference proceedings and has participated in several EU-funded projects with coordination and research roles. His current research interests include 5G networks, cloud/edge computing, and AI applied to these sectors. Currently,

he is an Associate Editor of IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT.

• • •