Empowering microservices: a deep dive into intelligent application component placement for optimal response time

Syed Mohsan Raza, Roberto Minerva, Barbara Martini, Noel Crespi

Empowering Microservices: A Deep Dive Into Intelligent Application Component Placement For Optimal Response Time

Syed Mohsan Raza[1*], Roberto Minerva[1], Barbara Martini[2], Noel Crespi[1]

[1]SAMOVAR, Institut Polytechnique de Paris, Palaiseau,91120, France.
[2] Universitas Mercatorum, Rome, Italy.

*Corresponding author(s). E-mail(s):
syed-mohsan_raza@telecom-sudparis.eu;
Contributing authors: roberto.minerva@telecom-sudparis.eu;
barbara.martini@unimercatorum.it ; noel.crespi@mines-telecom.fr;

Abstract

Microservice architecture offers a decentralized structure using componentization of large applications. This approach can be coupled with Edge computing principles: applications with stringent response time can benefit from different deployment options. However, it is crucial to gain profound insights into correlations between the deployment of distributed application components and the response time, especially from an application perspective. For correct placement decisions, it is important to evaluate the impact of small functions' placement and their interactions across the Edge-Cloud Continuum. This paper investigates the response time from an application perspective, considering the componentization using microservice architecture. Unlike the existing application placement approaches, we present extensive simulation results, illustrating the impact of service chains and marginally considered Application Programming Interface Gateways placement. Numerical evidence depicts that the design and placement of microservice-based applications could counter the common perception that Edge resources are always suitable for user-perceived response time. Further, we also present an experiment involving a componentized application and its optimized deployment in an actual testbed. Our findings and design guidelines inform effective component placement decisions while considering infrastructure constraints as well.

Keywords: Application Deployment, Edge-Cloud Continuum, Edge Computing, Microservice, Response Time

1 Introduction

The Edge-Cloud Continuum aims at the optimized organization of processing, storing, and communication capabilities into hierarchical segments of resources. Proximity of computational resources to final users can be exploited to improve the response time of large componentized applications by placing components close to their clients, minimizing communication needs. This organization defines a continuum of computation resources usable to satisfy different processing requirements. In the literature, these capabilities have been presented under different perspectives, like Edge Computing[1], Fog Computing[2], and Cloud Computing[3]. Their commonality is that the infrastructure allows for seamless integration of computational resources. This integration aims at exploiting resources with different kinds of capabilities to satisfy the diverse user application requirements in a timely fashion. Recently, microservices have emerged as a transformative architectural approach in software development that involves breaking down monolithic applications into smaller, reusable, and independent services. Each service is responsible for a specific functionality. The adoption of microservices aligns seamlessly with Edge Computing, offering a nimble and decentralized structure that complements the dynamic nature of distributed Edge-Cloud Continuum [4].

In this scenario, the optimal combination of componentization and deployment of microservices across different segments plays a key role in determining the overall application response time (i.e., the elapsed time since the user request is generated to when the response reaches the application or vice versa). How the application is split into components and where the components are deployed is relevant to the actual effectiveness of the application. Placement and aggregation of microservices are essential to support optimized interactions and exchanges of data among components to allow workflows to run in an effective way and accomplish the final goals of the entire application (e.g., timely address user demands) [5, 6]. The placement of large chunks of software in Edge nodes is not necessarily a good strategy if components are to be reused by other applications. Edge nodes could not be fully connected [7] with other nodes and this will increase the response time of the application. In addition, if some microservices are constantly reused by several applications, their placement should be optimized to favor general usage.

It is worth pointing out that microservices placement approach is different from minimizing the latency experienced by data flowing from/to users to/from the applications. A major difference is that a frequently requested data file can be cached, replicated, and moved closer to the final sinks to minimize the latency experienced by the user. Instead, functional requests (REST or RPC) between different software components that may have many relationships with each other should be dealt with "on time" and treated by specialized functions (i.e., software functions treating the user requests securely and concurrently). In addition, the requests deal with "functions" and related parameters. This means that user requests are to be processed on the basis

of actual and changing parameters in the request, which significantly increases the response time. Even "caching" interaction may result in an increase in total response due to the delay a request or message is waiting in a cache or queue before being processed. Migration could also be an option, but the quantity of data generated by a microservice as a response to a request could be minimized. The migration of the entire function elsewhere could be much longer and not effective for other users of the same application.

1.1 Motivation

To minimize application response time, there are various relevant approaches (e.g., [5, 8]), however, some limitations are related to the complexity of solutions in terms of algorithms and their implementations with respect to the practical deployment of real microservices-based applications (and related control chains). Many service providers (SPs) need to deploy them according to business requirements that could be counter-intuitive with respect to the goals of Edge-Cloud Continuum. These solutions and guidelines should be easily aligned with the work of developers. One of these examples is the Application Programming Interface Gateway (APIGTW) placement. It holds pivotal importance in large application deployment and is recognized as a significant coordination of microservices [9] aiming at load balancing. APIGTW also provides a unified interface to customers to all the changing functions and microservices developed over time for improving a large application. APIGTWs are points of control and administration that are extremely useful for SPs. With increasing application deployment demands, SPs need to deploy a large number of microservices and functions in stringently constrained environments in terms of Service Level Agreements (SLAs) and heterogeneous resources. In fact, administrative and business considerations could prevail over a pure algorithmic choice of deployment.

Edge computing's common perception favors placing the APIGTW in Edge nodes to facilitate latency-aware function invocations [10]. On other hand, Edge nodes are resource-limited and can not accommodate the seamless execution of hundreds and thousands of APIGTWs with hundreds of microservices [11, 12]. The coordinated microservices also need to keep close to APIGTWs which could not be convenient for supporting a limited number of users served by the specific Edge segment node.

In literature, it is trending to formulate multi-objective functions (for decreasing the overall cost and response time) and resolving them through suitable approaches, i.e., (mixed) integer linear programming adopted in [13]. Whilst, optimizing multiple Key Performance Indicators (KPIs) together, i.e., reducing response time, energy, and resource consumption, is mostly achieved through theoretical reasoning. This means that different parameters in computation or communication are heuristically assumed in such optimization while real network conditions could be more dynamic. Comparatively, it is also less evident in the practical scenarios because of per-service constraints on the segment's resource usage. During service execution, resources could not be dedicated to more than a certain limit to foster the computation of microservice functions [14], rather, a trade-off among the KPIs is required to maintain the QoS. For instance, such trade-offs are reported in [15–18].

A general view that should also be considered is related to the dynamic aspects of the usage of applications made out of microservices. In different moments of functions execution and for different users, specific control chains underlie and support the behavior of the application. Replication of microservices on edges, such as over different Mobile Edge Computing (MEC) edge nodes, can provide good response times. While it could be extremely costly or demanding in terms of control and management of the microservices.

1.2 Contribution And Paper Organization

Considering the mentioned problems above, we have the following contributions.

1. We focus on the intricate resource context in the Edge-Cloud Continuum, deploying microservice service chains of varying lengths, and APIGTW placement relative to microservice locations.
2. Response time problem is studied in detail from networking and application perspectives.
3. User-perceived response time of microservice chains is calculated using a simulated Edge-Cloud Continuum system. During simulation, APIGTWs are variably placed in Edge, Core, and Cloud segments, and also capabilities of segments are modified to find microservices chain placement impacts on response time. Such investigations are marginally considered in the Edge-Cloud Computing system.
4. A simple method for calculating the costs of different microservice chain deployments is proposed. Users can use the computed results and compare them to minimal acceptance criteria (e.g., maximum acceptable response time and costs) and choose a deployment configuration.
5. Simulation findings are discussed to inform the efficient early placement of microservices chains. The results favor placing microservices primarily in the Core segment and Edge, as well as aggregating full applications in Cloud, provided the perceived response time of Cloud deployment is not significantly higher than that of Edge and Core deployments.
6. To substantiate the results, we have deployed a microservice-based application in a Kubernetes cluster. The structure of a well-known test application (an application graph) was built and measured.
7. Based on the analysis of application architecture, and computed response times in simulation and real cluster, we consolidate the guidelines for the microservices placement in distributed resources of Edge-Cloud Computing.
8. Note that, the optimization of specific KPIs like CPU, memory, and energy consumption of microservices in different nodes is out of scope of this work. We generically refer to costs associated with the deployment of a microservice on a specific segment. .

This paper is structured as follows:

The existing related approaches for microservices placement are summarized in 2. Section 3 and 4 discuss the different delays introduced by systems and networks from an application perspective. These have implications on the overall application response

time. A model to study the application response time is presented in section 5. In section 6, various application deployment scenarios and consequent transport delays between the components deployed on different segment nodes are explained. Section 7 explains the simulation model and how the average response time of service chains is calculated. Next, section 8 explains a few possible strategies for microservices placement. Section 9 discusses the results related to optimized microservices placement and the insights derived from the analysis. Finally, conclusions and future research plans are explained in section 10, and 11.

2 State-of-the-Art

Deployment and placement of microservices in computing environments is a topic receiving a lot of interest from the research community at the Cloud and the Edge level [19]. Placement of software and microservices is also a major concern for Function as a Service approaches [20]. In this section, we present some relevant studies that consider the wide distributions of resources (i.e., generally in the Cloud or Fog and Edge segments) and the placement of microservices in these segments.

2.1 Application Optimization in Edge-Cloud

Under this category, several papers are focusing on the execution of applications made of microservices, mainly at the Edge (or Fog). The main goal is to optimize the allocation of Edge or Fog resources for hosting entire applications. Typically, the applications are related to the Internet of Things. Under this perspective, the majority of papers cope with the distributed and heterogeneous nature of the Edge and Fog nodes. They elaborate proposals and strategies considering the different performances of Fog nodes and network delays caused by connectivity between nodes in the Edge/Fog segment. The problem of optimizing coordinated microservices placement over different segments is only partially addressed.

In [5], authors present a static microservices placement strategy in a simulated network topology of Edge, Fog, and Cloud layer data centers. The proposed approach first places all the microservices of a service chain in the Edge node using a greedy approach. Afterwards, strongly coupled microservices (e.g., those that exchange a high number of messages) are collocated in the same node. Besides, this approach does not consider response time increase by communication of remaining microservices that are not collected and placed in different nodes.

In [7], authors discuss the implementation of Microservices Enabled Cellular Networks (DMCNs) in network Edge. Various Edge nodes offer heterogeneous computation capabilities with associated costs. A user request may invoke a chain of microservices placed at distinct and heterogeneous Edge nodes. These functions return the computed output to a central controller Edge node, which finally serves the end-user request.

Prediction helps in microservices deployment and node scheduling in EdgeCloud. In [8], authors proposed a method that predicts the upper bound of the response time of a microservice sharing processing capabilities. The proposed method anticipates

possible interactions between microservices, and then collected results are used to predict the best deployment.

In [21], authors characterized computational nodes by their profiles. An aggregator collects profile information of available resources and finds the best setting for microservices mapping (a constraint satisfaction problem). Afterward, software agents can deploy the microservices on Edge devices. The proposed architecture can support the easy deployment of microservices, but for the time being, it is abstractly defined. In [22], authors proposed a delay-aware dynamic microservice deployment and execution model. It essentially considers time-varying workloads and computational spikes in the Fog nodes. The model employs Reinforcement Learning (RL). It is a type of Machine Learning (ML) to directly learn in a dynamic environment, such as to decide the suitable target Fog node for microservice deployment. In [23], authors formulated Fog and Cloud nodes resource consumption optimization (minimization) by focusing on the per-service requirements. It uses batch placement, in which the microservices can be grouped together to meet stringent QoS requirements. To minimize the application response time, authors in [24] proposed a placement model in heterogeneous distributed Fog nodes. This model optimizes resource allocation using a Genetic Algorithm (GA) and calculates the multiple service chains composed of microservices. This approach was validated for average response time using Omnet++ simulation, and a microservices chain of smart city applications. In [25], a comprehensive study is conducted regarding the componentized application requirements and their different placement strategies in Fog and Edge nodes. The necessary dimensions for the service placement are discussed. These include whether the placement is instantiated from a central entity or distributed; whether the full application's requirement detail is available for adopting an offline placement strategy or otherwise online; and whether the placement strategy considers the dynamics of Fog, Edge, user (mobility), and application topology graph. The work does not explicitly mention the APIGTW implications of the aforementioned placement techniques. In [10], the authors proposed MicroFog, a framework designed for the efficient placement of microservices in federated Fog topologies and Cloud environments, as well as for response time-aware distribution of microservices. In [26], authors exploited that different chains of microservices compete to use reusable microservice instances; consequently, this competition prolongs the response time for users. To overcome this problem, the proposed approach called chain-oriented load balancing algorithm (COLBA) decides how many instances of microservices are fit for the chain's demands. In [27], a comprehensive survey was conducted to understand the application placement problem in Fog nodes. It shed light on the application architecture implications along with other criteria for placement. However, there is not any review of previous research work stated in this paper that has a partial or full focus on APIGTW placement implications.

In [28], a system is proposed to deploy distributed applications based on constraints identified by users. It is based on the extended functionalities of OpenStack, i.e., a Federation Flow Manager (OSFFM) which facilitates the deployment of distributed microservices over a federated infrastructure. An orchestration broker analyzes a Heat Orchestration Template (HOT) representing the user requirements. OSFFM utilizes an API to communicate with multiple Edge/Cloud orchestrators. In this way, it can

deploy microservices on the Edge of multiple OpenStack-managed systems. In [29], PerfSim is proposed to estimate the response time of microservice chains along with other KPIs. The authors describe how analyzing the behavior of service chains in a simulation environment can help in understanding and optimizing their execution on real systems.

Optimization of microservices placement in Kubernetes Clusters

Kubernetes[1] is the most adopted placement and orchestration solution for microservices in Edge, Core, and Cloud. It takes care of the efficient resource consumption (i.e., CPU) for each microservice execution; however, its default implementation disregards the microservices interaction models, response time, and computational node closeness. Various studies explore the strategies for optimal deployment strategies in the Kubernetes cluster and decreasing the impacts of default placement on application response time.

RuntimE Microservices Placement (REMaP) is proposed in [30] to address the problem of existing Kubernetes microservices placement in the Cloud. REMaP extends the Kubernetes scheduler to optimize the service replacement at runtime based on the service history (resource usage) and affinity. In [31], an optimization strategy for the placement of microservices in the Kubernetes cluster is proposed. The solution is based on the monitoring of network delays at the Edge of the network in order to make the orchestrator aware of varying delay situations of the infrastructure. However, the structure of the applications is not explicitly considered. In [32], authors extend the Kubernetes functions for more efficient placement and scheduling of applications in the Edge-Cloud Continuum. In [33], authors explored that a Kubernetes-managed application composed of microservices or an instance of microservices (e.g., replicas) can share common libraries or dependencies to achieve efficient resource usage. This resource sharing can overcome the resource competition problem in Kubernetes-managed applications. The method asserts constraints on response time. The required resources of microservice instances can exceed; however, the total required resources of diverse instances of a single microservice should not exceed the resources available in the underlying computational node. In [34], Kubesonde is a software solution integrated into Kubernetes aiming to find the microservices' connectivity (through their ports) and how the application-defined container access rules are applied. Although the major concern of this approach is microservices security; the lightweight probing mechanism in it is efficient for understanding the behavior of live microservices in Kubernetes clusters. Ge-kube is proposed in [35] to overcome the challenges of default Kubernetes implementations. The authors' highlight is that efficient application placement and scaling of the application components horizontally (increasing the number of containers) and vertically (allocating more computational resources for pods and containers) should adhere to the user's perspective of acceptable application response time. For this purpose, Ge-Kube integrates the elasticity and placement managers to take care of the application elastic scaling and placement in the geo-distributed heterogeneous resources. These managers proactively test the system and metrics like resource availability, delay between the worker nodes of cluster, and average response

[1] https://github.com/kubernetes/kubernetes

time of applications.Based on collected information a multi-objective optimization using RL enhance scaling and optimal placements.

Virtual Network Functions optimization through componentization (microservices)

Virtual Network Functions are seen as reusable components, providing network functionalities that applications can reuse. There is an interesting trend that studies how these functionalities can be decomposed and offered as microservices. Another aspect of this stream is how these VNFs impact the response time of applications and their structure, which is less considered and studied.

The authors in [36] explain the re-architecting of VNF using a microservice-based approach. It takes the example of a Service Function Chain (SFC) in the data center use case. This SFC is composed of VNFs like Wide Area Network (WAN) optimizer, application firewall, and Edge firewall. Authors decomposed the functionalities of these VNFs and found similar behavior of various functions for packet processing but they are vertically stacked in monolithic applications and can not modified for grasping the benefits of microservices architecture, i.e., scaling resources for single function.

Contain-ed is a system proposed in [37]. It aggregates the VNF components in the form of microservices. The communication relationship (number of messages exchanged) among the VNF components is referred to as affinity in the proposed work. In the proposed work, an affinity analytics engine finds the transaction frequency among the VNF and decides which NFV components should be aggregated, containerized, and deployed near the user. This aggregate of NFV components should respect the SFC latency-bound. In [38], authors proposed an approach for run-time migration of network applications (VNFs) deployed in the form of microservices. This work emphasized that after initial placement based on a greedy or best-fit approach, the gradual migration of microservices close to the request origin in data centers can decrease the global latency of applications. Nevertheless, the run-time migration strategy in advance needs to confirm a few metrics, essentially the impact of microservice migration on the overall application response time. In [39], the effects of the different placement of VNFs in the network and the impact on user-faced response time are considered. The goal is to minimize response time in a multi-Cloud scenario as an Integer Linear Programming optimization problem, which means caring for the SLAs. It models link delays and computational delays as queues and analyzes them from a statistical perspective.

Application Programming Interface Gateway (APIGTW) Placement

A few studies, for instance, [40] deeply investigate the coupling of APIGTW components, its implications, and different to this, how a decoupled architecture (pertaining to fewer interactions among components) could be adopted. In [41], authors identified the requirements and analyzed the formal description of commonly used 31 APIGTWs, including those used in Google and Netflix (i.e., Zuul API) platforms. Nevertheless, the requirement of distributed placement is not considered in this work. In [42], a security agent is implemented for authenticating the user in Edge computing platforms. The other research works, like [43–45], mostly focus on the architecture and benefits APIGTW offers in terms of load balancing, security, authentications, proactive content caching for microservices, fault tolerance on gateway nodes and enabling proxy

for microservices. However, none of them were dedicated to informing the placement's implication of APIGTW on the response time of application in Edge-Cloud Continuum. Therefore, our findings and proposed approach are different from the existing studies to timely inform the importance of the marginally considered research gap.

3 Response Time Problem from the Network Perspective

The Edge-Cloud Continuum within the network infrastructure is structured hierarchically, comprising distinct segments or administrative domains that serve specific roles, infrastructure functions, and business models. These segments are operated by various entities, including User/Enterprise, Communication Service Provider (CSP), Internet Peering Provider (IXP), and SP. Each of these actors plays a unique role in the overall organization and operation of the Edge-Cloud Continuum. Mimicking Content Delivery Networks (CDN), the usual approach used to reduce the latency of applications is to deploy mini data centers at the Edge of the network and make them available to deploy SPs' applications. As for CDNs, different configurations are possible: Edge nodes can be in the CSP network (e.g., Telco CDN). Similarly, SPs can deploy a part of their services and functions at the Edge using the infrastructure made available by the CSP; or within the infrastructure of IXPs (IXP CDN) [46]. In the case of IXP CDN, the Edge nodes will be further away, but the IXPs allow the SPs to deploy their own hardware and software (e.g., Netflix Open Connect). The CSP can also equip the Core network segment with processing capabilities (Core data centers).

CSPs are exploring Edge deployment of applications to notably decrease response times, aligning with CDN practices. However, the focus tends to be on the overall application rather than the nuanced architecture and interactions among its components during deployment. CSPs commonly view response time optimization as a matter of physical proximity, believing that placing resources closer to end-users reduces roundtrip delays for interactions. On top of this, it should be noted that the CSPs are deploying Virtual Network Functions (VNFs) in their network infrastructure, according to the Network Function Virtualization paradigm [47]. This means that virtualized resources are used to deploy network functionalities other than to execute application components. Application developers are thus capable of integrating distributed VNFs within "chains of control" that characterize the behavior of the entire application [48].

The impact on response time induced by the different deployment options of the (componentized) application and its chain of interactions with VNFs is usually neglected. For instance, an application deployed at the Edge requesting a security-related VNF (e.g., a firewall deployed in the Core of the network) should consider the delay due to the time needed to send request messages to the NFV and receive the response.

Within this context, specific aspects to be addressed from a network perspective are explained in the following sections.

3.1 Capabilities and Topologies of Edge and Core Nodes

Edge data centers are designed to deploy functions and/or run services for a limited number of users connected to a single Edge node. The infrastructure deployed at Edge and Core nodes are typically computationally powerful [49], [50], [51], but will not be comparable to gigantic data centers. The functions or software solutions deployed in the Edge to operate the infrastructure should be specialized. For instance, the user-defined orchestration solutions should not have a higher CPU or memory footprint in Edge nodes. On the other hand, when deploying microservices, Edge node connectivity must be considered. If Edge nodes are not directly connected, then the placement of functions on near nodes can have a slightly longer response time. If the Edge nodes are connected using Core nodes (with Core nodes typically serving more Edge nodes), then the placement of functions in Core nodes is preferable for placement, rather than a farther-away Edge node. The type of functions to be deployed at the Edge is another important choice to deal with. They should be characterized by a high level of reusability (many users will invoke them concurrently) and by a limited span (i.e., they should limit the need to interact with other functions deployed elsewhere in the Edge-Cloud Continuum).

Only a fraction of Internet Cloud applications and services can be fully deployed at the Edge. The SPs have developed over the years a large number of components and functions that are adequately placed in the Cloud to serve users' requests [26]. The combination of load balancing options and the APIGTW approach for managing large componentized applications is in place to guarantee the right response time for many applications. Some functions and related microservices, e.g., those dealing with the retrieval of information, are better performed when co-located close to the databases. This is also true for largely distributed databases. They are storing data within the Cloud infrastructure and duplicating some instances at the Edge when requested.

Another concern impacting the response time of apps is how the resources of Edge data centers owned and operated by CSPs will be allocated to SPs. Edge computing resources can be allocated to SPs "on-demand" and in competition with other stakeholders, or according to general contracts between CSP and the SP. These contracts guarantee processing capabilities. Computing resources can then be permanently allocated to a specific SP that will decide when and what to instantiate. It can also be dynamically allocated to different SPs when resources are assigned based on competing and varying requests for specific services of different SPs. In all these cases, the satisfaction of response time requirements of the application itself is either considered by the SP before requesting resource allocation or is neglected.

3.2 Transport and Deployment Induced Response Time

In largely distributed applications, messages and software invocations are to be passed from one subsystem to the other (with some delay due to the transport latency) and then processed by the operating system and virtualization layer before being passed to the microservice/functions. Correspondingly, delays are introduced that are due to communication delays between the involved subsystems and processing delays within the subsystems themselves [52].

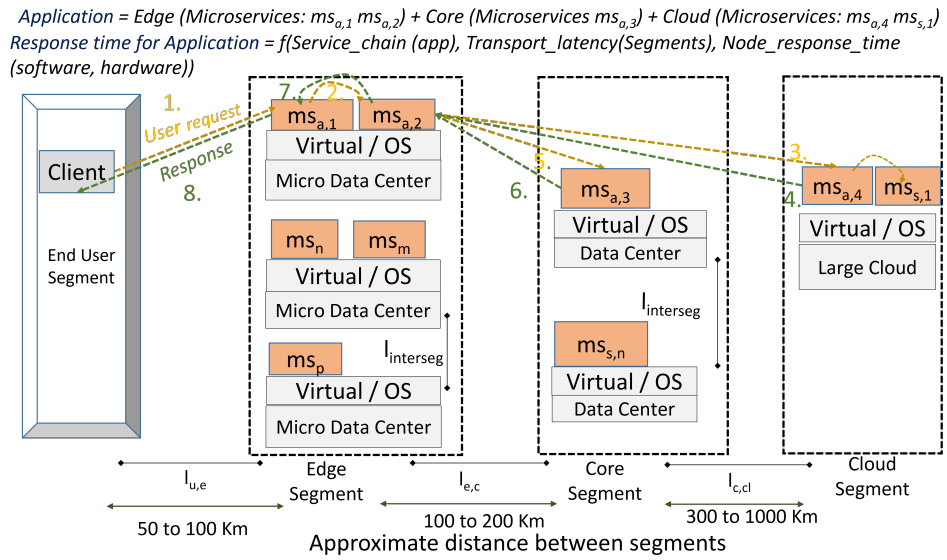

Fig. 1 A presentation of application request and response time from connected segments

Figure 1 presents a scenario of a distributed application composed of five microservices and deployed across the Edge, Core, and Cloud segments while processing a user request to generate a response back to the user. The client invokes the first microservices of the chain $\mu_{s_{a,1}}$. The microservice $\mu_{s_{a,2}}$ in Edge, further invoke the microservices $\mu_{s_{a,3}}$ in Core data center and $\mu_{s_{a,4}}$, $\mu_{s_{s,1}}$ in Cloud data center. This chain execution incurs delays due to microservices execution time, communication lags between segments and computation nodes processing. The aggregate of these delays impacts the overall application response time.

As for communication delay and as calculated in [53], the network roundtrip time for a message from the customer device to a responding server (e.g., for a web service, a REST call, or requests enabled by HTTP protocol) can vary from 1.6 ms for servers located at a distance of less than 100 km to up 96 ms for multi-continental distances. These values can be acceptable for some applications but not for more real-time ones. The correct placement of components is then an important issue to improve the responsiveness of applications.

3.2.1 Node Delay

The computation node delay is due to the aggregate effect of hardware capabilities and software complexity. In particular, the software delay includes: 1) the operating system (e.g., scheduling of jobs); 2) the Remote Procedure Call environment (encoding and decoding of the parameters); 3) the virtualization environment (containerized microservices share the single operating system kernel), 4) orchestration solutions managing the microservices life cycle (like Kubernetes); and 5) add-on software (i,e, Istio service mesh) agents or proxies in microservices for distributed application tracing and traffic management in microservices.

4 Response Time Problem From the Application Perspective

In the Edge-Cloud Continuum, response time should account for the wide distribution of software components across the infrastructure and how they interact and synchronize their execution. It should also consider the delays introduced by systems' mechanisms and software for reusing the same components deployed in multi-application environments. The systems on which components are deployed introduce delays themselves, as defined in section 3.2.1. Virtualization provides flexibility for the deployment of several applications and components in reusable systems, but it introduces additional delay for: 1) mechanisms supporting the sharing and allocation of server resources; and 2) additional processing overhead to support virtualization. There is a continuous evolution towards developing efficient systems with lower overhead for resource virtualization and orchestration of resources [54], e.g., Dockers and Kubernetes.

These systems support microservice architectures [6] and applications. Microservice architecture is based on the decomposition of large applications into a set of small, reusable, distributed, efficient, and optimized small modules, each one providing a well-defined function. Microservices are flexible means to promote the reuse and composition of software. The single function can be easily updated without a massive impact on other functions of applications. The programmability is guaranteed by means of the definition of well-formed interfaces for accessing the function offered by each individual microservice. Each microservice can be accessed using an Application Programming Interface (API). Microservices are packaged in units and deployed within different execution environments as containers or pods [55]. Best practices for microservices [56] suggest packing them individually or in a small group of a few strongly related units. Packaged microservices can be deployed on different segments depending on the availability of resources. The communication between these packages of microservices (within the same execution environment or in different segments) should be optimized with respect to the response time of applications. The term microservice, in this paper, refers to a simple deployment unit comprising one or a few fully correlated functions. A Kubernetes pod is the smallest deployment unit. It may have a single or more deployed containers that jointly offer the functions of a microservice. These containers share a single namespace, host operating system, underlying computational node, and IP address.

For applications, response time is the aggregation of different delays introduced by software components, systems, and network segment communication. Throughout the paper, the term delay is used to avoid confusion with the term latency, typical of telecommunications. Here, the focus is on how reactive the different systems are in order to reduce the response time of the applications by optimizing the organization and arrangement of its components.

Message delay is the time elapsed between the sending of a message from a software module and the receipt of it by another one located in a remote node. It includes the transport delay and the time needed to encapsulate and de-encapsulate the remote procedure call methods and parameters into a message. System delay is introduced

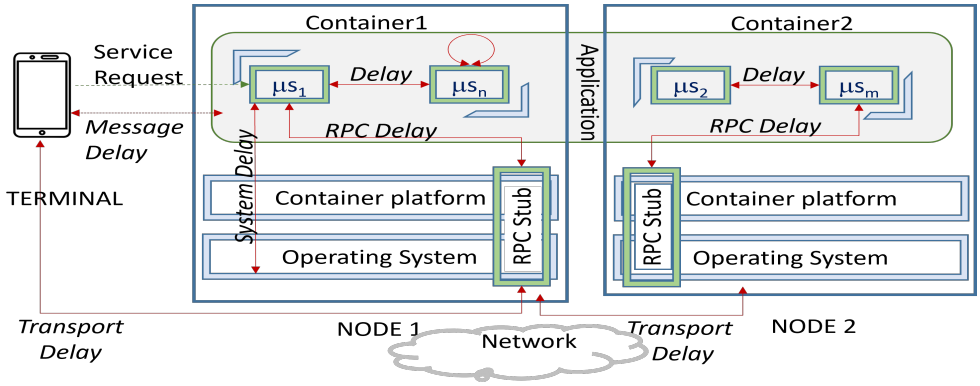

Fig. 2 Different types of Delays contributing to overall response time in a simple distributed system

by the software infrastructure supporting the execution of software modules (e.g., operating systems and virtualization mechanisms). Figure 2 represents a Terminal requesting a service offered by a networked application. The application is composed of microservices deployed in separate packages over two different nodes. Individual contributions to overall response time (different types of delay) are highlighted.

The reasons for a flexible distribution of functionalities (and related microservices) over an Edge-Cloud Continuum may vary. Some functions (e.g., those related to the Internet of Things) may reduce the volume of data to send from the source to the final destination, or they can require the application of AI techniques to identify patterns or identify objects. These functions should be characterized by a reduced number of interactions with functions deployed in other nodes or segments. From another perspective, some functions may need specific processing or have direct access to databases. Some microservices are to be executed in the SP environment for security, management, or consistency reasons. More cases depend on the type of applications as well as the requirements and constraints of the SP. Depending on the type of function, the number of expected interactions with others, and other constraints of ownership, microservices could be conveniently deployed at the Edge.

As a consequence of this flexibility, the Edge node resources need to be orchestrated within the whole Edge-Cloud Continuum to identify the best deployment and execution options from the application perspective. In a highly dynamic system, some Edge nodes may be overloaded, so new microservices need to be deployed and executed in other nodes, or a few old microservices need to be migrated to other Edge nodes for optimization reasons. In these cases, delays may arise due to the distribution of the microservices comprising the application. Continuous interactions between microservices deployed in different nodes can significantly contribute to the delay of the application in providing results.

Figure 3 represents different cases of delays due to the positioning of microservices. If the End segment is directly connected to all the Edge nodes in the segment, the deployment of a microservice can exploit this full connectedness. The client applications could interact with microservices deployed anywhere in the Edge Segment with minimal fluctuations in delay. The delay between the user segment and Edge nodes,

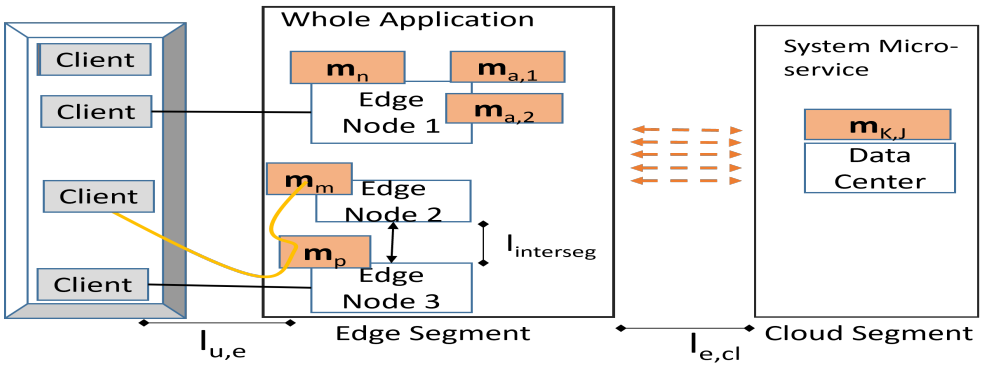

Fig. 3 Migrating microservices at the Edge and frequently used Microservices

represented by the value of $\ell_{u,e}$, will be essentially the same for all the interactions. If the End User Segment is not fully connected to the Edge Segment, the deployment or the migration of a microservice to an Edge node not directly connected to the user segment may have an impact on the total delay. It should consider the inter-segment delay between Edge nodes ($l_{interseg}$) and the delay between the user segment and the Edge segment ($\ell_{u,e}$)). In a hierarchical network (e.g., in the Mobile Edge Computing approach), the users are connected to a single Edge node. Due to the lack of direct connection, the migration of microservices from one Edge node to another one involves a longer path (through a Core segment). Interactions between microservices residing on different Edge nodes managed by the same node of the Core Segment suffer a longer delay to pass through the different segments, and this can have an impact on the response time of applications. Figure 3 shows also the case in which two microservices placed in different segments have multiple interactions. In this case, the delay needs to consider the value of the (double) times taken by the interaction between the Edge and the Core segments (represented as $\ell_{e,cl}$).

As shown in Figure 3, the case of minimal delay, e.g., when all the application's components are in the same Edge node 1, holds. The delay due to the communication between nodes is null and only additional (smaller) delays introduced by the processing power of Edge node and the optimization of virtualization and communication components are to be counted. If one of these microservices is placed in other nodes, additional delay is introduced by transmitting the data and extracting the messages at the remote microservices system. This distribution of microservices (e.g., in another Edge node, in the Core, or the Cloud) results, then, in increased delay. For instance, the microservice operating on Edge node 2 and interacting with the microservice on the Cloud Segment experience a long delay for each invocation. This results in a long response time for the client requesting functionalities in the End User Segment.

The current strategies of orchestration favor the optimization of resource allocation at the system level, while the optimization of the application is still left to the application designer [57] by means of description languages or JSON data. Also in the serverless domain (i.e., Functions as a Service, FaaS), the focus remains on the system optimization, and the analysis of the optimization and orchestration is not yet taking

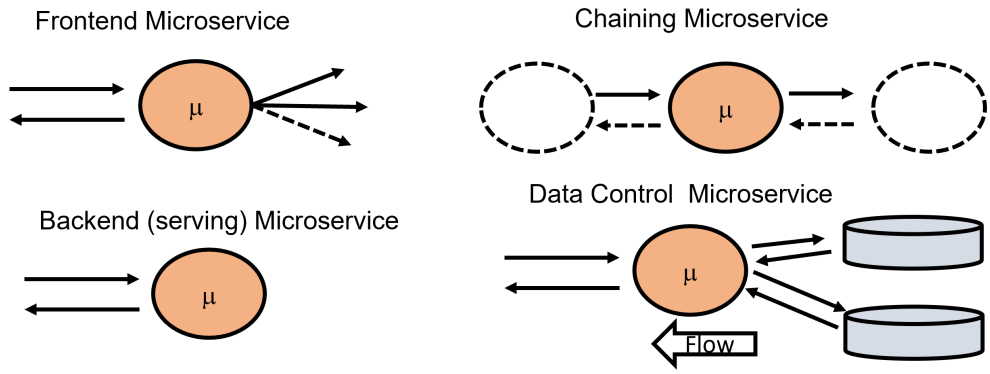

Fig. 4 Different types of Microservices and their interactions

into consideration the deployment in an Edge-Cloud Continuum. One of the goals of serverless orchestration is to reduce the time and complexity of optimizing how applications and their functions should be executed in the Cloud [58]. There is a need to analyze from the application perspective what the best deployment options are aiming for in terms of optimizing the response time (at least for the normal execution flow).

4.1 Types of Microservices Interactions

In a microservice architecture, the different functions are tied together through API calls. The flux of calls will depend on the general structure of the application (the intended behavior and goal) and the specific parameters provided by the user invocation (e.g., exceptions, border cases, or regular behavior). The microservices create a sort of execution chain to perform computational tasks of the application and return values to the user. The structure of programs can be represented as a Direct Acyclic Graph (DAG). Applications made out of microservices invocations are also depicted as graph [59, 60]. Such a representation can be used to understand how the different microservices interact and what kinds of dependencies exist between them. At a more basic level, there is the possibility to classify typical microservice behavior in terms of interactions with others [61].

Figure 4 presents four basic types of microservices. Frontend is a microservice that receives several invocations from the clients and can coordinate the execution of "depending" microservices. On the other side, Backend microservice can be seen as a function that processes the input and returns a value to the invoker. A specialization of this is the Data Control microservice, whose goal is to perform calculations based on data stored in an associated database (or to control the streaming of data from the content database). The Chaining microservice receives invocations, performs some calculations, invokes other functions, waits for their completion, and then updates the results and returns the value to the invoker. This is typically an intermediate node in a chain of control.

Frontend microservice type is a simplified representation of the API Gateway [62]. This type of functionality is characterized by a large number of interactions with other functions, and its placement in different segments can have a relevant impact

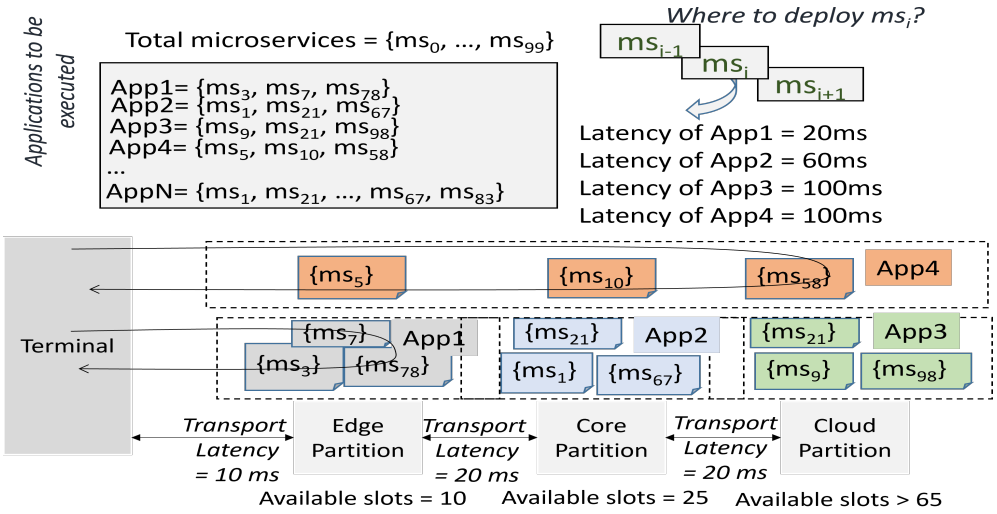

Fig. 5 Edge−Cloud Continuum segments and Service Chains deployment on available slots

on the response time of the application. These different types of microservices can have an impact on how the control chains work, and a wrong placement of them in a segment could have a relevant impact on the total response time. For instance, placing a chaining microservice in a badly connected node could not be effective, as well, placing a FrontEnd microservice too far from frequently invoked BackEnd microservices could not be efficient. The structure, the type, and the placement of microservices in a chain of control should be considered to optimize the overall response time.

5 A Model For Studying the Application Response Time Problem

The microservices allocation problem is studied using a simple model. It addresses the mechanisms and effects of a wide distribution of microservices on a segmented processing and communication infrastructure. The model helps in analyzing deployment and allocation scenarios to determine transport delay and help in the minimization of response time. Figure 5 represents a high-level model of the Edge-Cloud Continuum comprising segments, i.e., sets of nodes such as data centers capable of hosting and executing microservices. They have the property of being placed in well-geographically determined positions concerning the users served. In the Figure, Edge, Core, and Cloud segments are depicted, but the segments could also refer to other geographical separations (e.g., regional, national, and international segments made out of different size data centers about the associated providers). The nodes of a segment exhibit similar characteristics in terms of processing capabilities (e.g., processing time for dealing with RPC Calls or the same type of virtualization capabilities) and delay in message passing from one segment to the other. The time for passing a message from Edge to

Cloud segment is additive, i.e., the delay is the sum of the delay from Edge to Core and Core to Cloud. The Figure also presents a set of microservices (from 0 to 99), some of them are already deployed in the nodes. Applications are represented as a simple chain of microservices invocation. The End User Segment represents the set of endpoints connected to the Edge Segment. They are the clients of the applications distributed in the Edge-Cloud Continuum. Communication between segments is subject to transport delay (in this case, strongly related to network latency). The figure shows how different deployments can introduce different response times for invoking Clients. The initial basic assumptions (derived from industry analysis [49, 63], current deployments of products like EdgePresence, and the current definition of the MEC architecture [64]) are:

-The Edge segment comprises a data center (an Edge node) supporting a limited set of microservices. In the case of two or more Edge nodes, the assumption is that they are not directly connected, but are linked through the Core Segment. This is to reflect the current organization of Mobile Edge as it is taking shape in industry of Mobile Communication. However, if the Edge nodes are directly connected, the transport delay inter-segment between nodes should be considered.

-A Core Segment comprises a data center. Inter-Core delay, i.e., the time for forwarding packets between two data centers in the Core partition, is not considered. The assumption is that the serving Core data center is capable of hosting the microservices that interact with the directly related Edge ones. Due to the hierarchical structure of the communications network, one Edge node is connected to the Core node. Core node can be directly connected between them, but there will be transport delays for the communications.

-A Cloud segment represents a large data center with a high capability to host microservices.

-An Edge segment has fewer capabilities than a Core one and even fewer than a Cloud segment in terms of processing and storage power. The hosting capabilities, i.e., the number of slots for hosting microservices, are less than the total capacity of the Core and by far lower than that of the Cloud.

-System, Virtualization, Remote Procedure Call, RPC, and local delays are disregarded because the focus is to determine the impact of the distribution of microservices on the global transport delay, in relation to network latency. Communication delay of two microservices in the same partition is set to zero. If an application is entirely deployed in a single segment, the system and software delays will not be included.

-The execution time is set to zero for each microservice, regardless of the segment in which it is executed. In reality, the execution-time delay is not zero, but it is a fraction of the transport delay.

-There are different transport delays between segments. They are cumulative, i.e., the transport delay between two segments is the sum of the transport delays of the traversed segments. In the model, the different values of delays between the segments are programmable and hence can be adapted to the real values monitored on the field.

-Applications are defined as service chains, i.e., a sequence of invocations between microservices that compose an application. The service chain is triggered by a request from a Client in the End User Segment. Microservices are executed in two modalities:

a cascade, where one microservice invokes the next one; or controlled, in which a controlling component (a gateway or a microservice) invokes microservices in the chain one at a time. Parallel execution of microservices or a combination of cascaded and controlled execution will be added further on. The controlled mode is particularly important because, currently, the SPs are using API Gateways to control how users are accessing, using, and taking advantage of the applications offered on the market. API Gateways are means to offer a stable API to users and keep flexibility in backend to improve, change, add, and increase the functions and parameters of microservices. The placement of API Gateways is then extremely relevant for load balancing the requests and for offering a proper response time to end users.

6 Application Deployment Scenarios and Transport Delay

Figure 5 depicts the organization and execution pattern of four applications. Applications are considered chains of microservices. The figure considers a total capacity of 100 microservices. For the time being, they are unique microservices without replication of some of them. This makes the model simpler at an initial stage. However, duplicated microservices could have the same name, but each replica will have a unique identifier. So the model will still be valid. The distribution of microservices in the different segments is random and is limited by the capacity (in terms of slots) of the individual segment. For each application, coordination of set of microservices is needed. The composition of service chains is unconstrained, and so the microservices can be invoked in any order. For this reason, in the model, the service chains are built randomly. In a service chain of length 5, only 5 microservices will be selected in any casual order. However, in reality, there may be ordering constraints, e.g., identification and authorization microservices are most likely used as the initial functions of applications, while a backend microservice chains are randomly built to keep the modeling as generic as possible.

Simple initial scenarios model how deploying microservice chains affects application response time (full roundtrip from request to reply). For example, microservice placement and transport delay impact response time based on the deployment and execution of the service chain over several nodes. Figure 5 depicts four applications modeled as chains containing three different microservices each. App1 is fully deployed at the Edge, App2 in the Core, App3 in the Cloud, and App4 is deployed on three different partitions. The transport delay between end user and Edge segments is set at 10 ms, while between Edge and Core, and Core to Cloud is set to 20 ms for each packet transmission. A total transport latency of 50 ms is assumed for sending a message from a client in the End User segment to the Cloud one.

App1 has a quick response time (a total of 20 ms): the Client sends an invocation to the service chain residing in the Edge node (10 ms of latency), the App1 is fully executed at the Edge and the result is returned to the Client (additional 10 ms of delay). App2 is similar to App1, but the transport delay to reach the Core nodes is greater (10ms + 20ms), resulting in a round-trip delay of 60ms. App3 and App4 have very different distributions of functionalities, but, under the conditions and constraints

discussed, they have the same transport delay, i.e., 100 ms. The usage of valuable slots in the Edge and Core partitions for App4 does not help to decrease the total response time of the execution of the service chain.

The deployment of even one microservice in the Cloud segment, while others in Edge or Core segment can affect the total response time of the application. It can also make the deployment of other microservices of the chain sub-optimal which are already deployed in scarce slots of the partitions. This may hold for long chains of cascaded microservices, in which the probability of at least one microservice being deployed in the Cloud increases. Nevertheless, software architectural choices in microservices-based applications can have an impact on global transport delays. The APIGTW [43], is a coordination point for orchestrating and using the different microservices comprising an application. APIGTWs are enablers for SPs because they expose in a controlled manner.They are also used for load balancing, ensuring how applications (and their components) offer functionalities to end users. APIGTWs support the orchestration of applications in different data centers of the SPs. They are also cause of additional delay in Microservices Architecture [65]. For these reasons, the ability to guarantee the SP their full control as well as correct placement of them are key factors for deciding if and where to place APIGTW in an Edge-Cloud Continuum infrastructure.

API Gateways impact the entire service chain execution, and their placement in one of the segments of the Edge-Cloud Continuum should be carefully considered. Deployment of these functions at the Edge or even at the Core may raise ownership and control issues for SPs. The deployment of API Gateways in different segments and the relations with co-located or remote microservices have been considered and represented by the Edge-Cloud model that was described before. The control relationships and the invocation between microservices have been analyzed to understand the Edge-Cloud Continuum delay for largely distributed applications.

Different service chains comprising an APIGTW and three microservices have been considered. The APIGTWs have been placed in different segments, and the related microservices have been placed according to different meaningful configurations in order to illustrate the consequences of the placement of components with respect to the interaction with the APIGTW. The colors assigned to the APIGTWs and the microservices reflect their placement within a segment. The delay associated with the interaction between components in the same segment is assumed to be 0, interactions between components in different segments suffer the transport delays between segments. The delay between non-adjacent segments is the sum of the delay for reaching the desired segment passing through intermediate segments. The response time is calculated by summing up the time needed to reach the components and to get a response (i.e., twice the delay between interactions). In this way, for each application, a calculation in terms of response time due to the interactions between the different components placed in each particular configuration is made. Among different possibilities, five scenarios are depicted in Figure 6:

App1: An API Gateway, APIGTW1, is deployed at the Edge. The delay is equal to the time to reach the API Gateway interface at the Edge and time to receive an answer, and the time for the APIGTW to execute the calls to its microservices (time

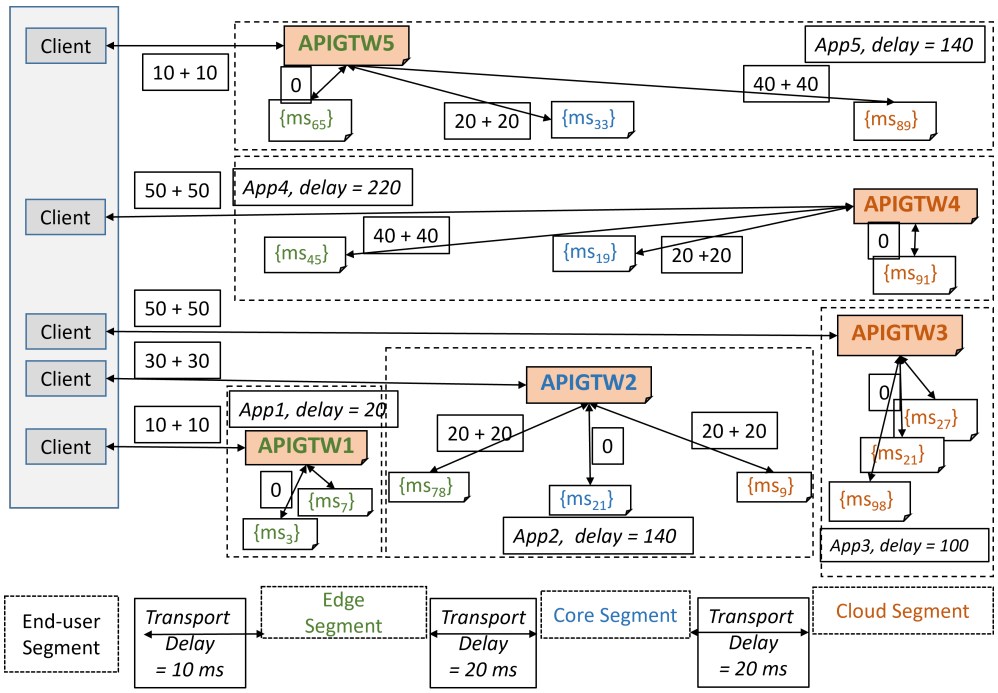

Fig. 6 Deployment scenarios for API Gateways and related transport delay

assumed to be 0). The total response time for the application (based on the transport delay) is then 20 ms.

App2: An API Gateway, APIGTW2, is deployed in the Core segment. The transport delay to and from it is 30 ms (for a roundtrip of 60 ms). Three microservices are deployed, one for each segment. Two of them (deployed in other partitions) add a total roundtrip of 40ms. The total Response time for this configuration (due only to transport delay) is 140 ms. A large part of the Response Time is due to the need to interact with microservices deployed on other partitions (an effect of the transport delay).

App3: An API Gateway, APIGTW3, and all the needed microservices are in the Cloud. The transport delay is related to the roundtrip delay of the invocations between the client in the End User Segment and the Cloud Segment hosting the API Gateway. The total response time is 100 ms (entirely due to the transport delay for reaching the far-away Cloud Segment and its applications).

App4: the API Gateway APIGTW4 is deployed in the Cloud and each segment hosts one microservice of the control chain. This scenario pays heavily in terms of transport delay for the back-and-forth interactions between microservices running on different segments. The total response time is 220 ms mostly due to a very ineffective placement of components. Even if one component is in the valuable Edge Segment, its placement there does not help, and even increases the response time.

App5: the API Gateway, APIGTW5, is deployed at the Edge, but the three microservices are distributed on each segment. The total Response time is 140 ms.

Also, in this case, two components are placed in the valuable Edge segment, but the response time is not optimal.

The best scenario is when the APIGTW1 and all the needed microservices are co-located at the Edge. However, this means that the API gateways of SPs should be moved to the Edge. However. such migrations raise the intentions that how many API Gateways (plus all the used microservices) can be supported at each Edge node? What are the implications for the SPs in terms of replication of API Gateways for each Edge node to cover? These considerations can make the deployment of API Gateways at the Edge problematic for many SPs. The second best scenario is to deploy the API Gateway and all the microservices in the Cloud. The difference in terms of response time (induced only by transport delay) is relevant (over 80 ms of difference), however, the benefits for the SPs could be considerable: the total control of service provision is on their centralized infrastructure, and the SP maintains the ability to exploit the richer capabilities of well-known infrastructures. The other scenarios have pros and cons that should be evaluated by the application developers and SPs case-by-case depending on specific constraints and requirements. These scenarios, show that the topology of placement of service chains of microservices has a relevant impact on the response time of the specific application. Focusing on the optimization of the allocation of resources in the Edge-Cloud Continuum should be a trade-off between the optimization of the allocation of resources in the different segments together with the needed optimization of the application based on its topology (or graph).

Two shreds of evidence emerged from the analysis of these scenarios:

Aggregate on the lowest delay segment if the entire service chain fits: The allocation of a single or a small number of microservices in the lower delay segment does not always minimize the global response time. If the entire service chain can fit, it should be placed in the segment with the lowest possible delay. This obviously depends on the current hosting capacity of segments on the Edge-Cloud Continuum.

Aggregate on the highest delay segment: if a service chain is mainly placed in a lower delay segment, but it has two or more invocations to microservices residing on a higher delay segment, then place the entire service chain in the higher delay segment to decrease the total transport delay. This holds especially when the difference between the delay of adjacent segments is large.

These considerations can be used as initial steps for placing and orchestrating the deployment from the application perspective. They can also be used when migration of microservices from one segment to the other should take place. Migration of single microservices (of different service chains) can have a limited impact on response time, while migration of two or more microservices can have a greater impact. For instance, the study [5] attempts to allocate microservices in an optimized way in the Edge-Cloud Continuum. The idea of co-locating microservices in segments to reduce global transport latency is an emerging concept.

7 The Simulation

A model of chained microservices based on the assumptions discussed earlier has been implemented on the OMNET++ simulator [66]. It has been developed to study generic

placement scenarios for service chains of different lengths. The simulation focuses on the analysis of transport delays introduced by the decomposition of applications into service chains and their deployment on available segments. It offers the possibility to parameterize the values of service chain length, distribution of slots in terms of percentages of each segment, and different delays between segments. The simulations focus on the behavior of unconstrained microservice chains and of API Gateways at the Edge, Core, and Cloud. Segment capabilities and segment-to-segment transport delay are parametrized so that different scenarios can be adjusted to actual or prospective measures. The hosting capacities of Edge, Core, and Cloud data centers are also parametrized. It is assumed that the number of slots available in the Edge-Cloud Continuum is sufficient to host the entire set of needed microservices.

The parametrization of the simulation takes place in the initialization phase and it is supported by a file that contains the chosen parameters that characterize the session of simulation. An example of the parametrization of the initialization file for a specific simulation is presented in Figure 7. The initialization file is composed of various user-defined parameters. The values are 25% Edge slots, 35% Core ones, and 45% at the Cloud. The service chain length is set to 3. The actual microservices percentage parameter (*MicroServices_percentage_**) reflects the distribution of the microservices in a segment. The per-service chain length is the number of coordinating microservices to complete a user request. The placement parameter (*Where_to_put_the_Microservices*) is to decide whether all the microservices should be deployed in a single segment or they should be distributed. If they are in a single segment, then the full microservice percentage (100%) will be deployed in single-segment computation nodes. The number of the sequence parameter (*User_defined_number_**) specifies the total number of microservice chains that should be traversed during the execution time of the single simulation. The reason to explicitly mention this parameter is that the number of possible chains using the binomial coefficient formula is much larger. So invoking all the microservices composed in all possible chains in a single simulation period can be highly compute-intensive for the underlying machine. The service API placement parameter indicates in which segment to place an APIGTW. These segments are the Edge, Core, or Cloud. After performing the computation, microservices coordinate with the API Gateway and send the response to the client application (terminal).

The simulator employs:

1. the Fisher-Yates shuffle algorithm to randomly distribute the initial vector of the microservices;
2. Dijkstra's algorithm determines the best route to reach from one end to the other end of the service chain. It returns the list of intermediate nodes to be traversed to reach the desired microservice;
3. the binomial coefficient formula ${}^{n}C_{r} = n!/(r!(n-r)!)$ generates the unique sequences of a given length of functions to be invoked on user request. In this formulation, n presents the total number of microservices for the deployment while r denotes the required chain length. Note that, this allocation of microservices to segments is random and it should not be considered as an allocation algorithm.

```
[Configurations]
network = MicroServices_Simulation.Topology
**.sendIaTime = exponential(1s)
**.MicroServices_percentage_edge_servers =  25
**.MicroServices_percentage_core_servers = 30
**.MicroServices_percentage_cloud_servers = 45
**.number_of_the_services = 100    // micro services
**.per_service_chain_length_User_defined = 3
**.User_defined_number_of_chain
_out_of_total_Unique_Seq_end = 16000
**.Where_to_put_the_Microservices= "Distributed“
**.Where_to_put_the_Service_API = "Cloud_Interface"
//**.Where_to_put_the_Service_API = "Edge_Interface"
//**.Where_to_put_the_Service_API = "Core_Interface"
//**.Where_to_put_the_Service_API = "Mobile_Interface"
```

Fig. 7 Simulation parameters

The goal of the simulation is to help determine the patterns and behaviors of randomly placed service chains. The process is as follows: in the first step, the 100 microservices are randomly allocated to the available segments; in the second step, a subset of all the possible permutations of microservices in service chains of a given length is selected; then the simulation measures the transport delay for each of the selected service chains. In the simulation, the microservices have been distributed on the different segments in such a way as to fully use the slots in the Edge or Cloud segments. This means that, from a resource allocation perspective, all the resources are fully allocated and the allocation can be considered optimal because all the microservices can be executed. Changing the microservices placement does not improve (or decrease) the effective capabilities of the Edge-Cloud Continuum total capability. From an application perspective, instead, different placements of microservices can have impacts on the total response time.

The cycle of simulation follows the steps below:

1. The simulator randomly places microservices in the segments.
2. It generates a large set of service chains (each represented by a vector of microservices). The selected service chains are limited here to 16000 through the user-defined parameter in order to cope with the exponential nature of the problem. Note that serializing the 16000 service chains adequately captures the dynamicity of the chains, and geographically varying deployment. Moreover,

Table 1 Response time calculation based on service chain deployment, length, and partitions capabilities

index	Chain Length	Slot Distribution in Edge, Core, Cloud	Segment to Segment Delay	Edge GTW	Core GTW	Cloud GTW
1	3	5%, 50%, 45%	[1ms, 5ms, 5ms]	0.421s	0.025s	0.428s
2	3	5%, 50%, 45%	[1ms, 10ms, 1000ms]	2.357s	2.323s	5.831s
3	3	25%, 30%, 45%	[1ms, 5ms, 5ms]	0.035s	0.031s	0.048s
4	3	25%, 30%, 45%	[1ms, 10ms, 1000ms]	2.344s	2.335s	5.753s
5	3	50%, 20%, 30%	[1ms, 5ms, 5ms]	0.023s	0.034s	0.061s
6	3	50%, 20%, 30%	[1ms, 10ms, 1000ms]	1.352s	1.375s	6.060s
7	4	5%, 50%, 45%	[1ms, 5ms, 5ms]	0.061s	0.034s	0.054s
8	4	5%, 50%, 45%	[1ms, 10ms, 1000ms]	4.287s	4.625s	7.036s
9	4	25%, 30%, 45%	[1ms, 5ms, 5ms]	0.058s	0.042s	0.047s
10	4	25%, 30%, 45%	[1ms, 10ms, 1000ms]	4.668s	4.636s	5.591s
11	4	50%, 20%, 30%	[1ms, 5ms, 5ms]	0.045s	0.045s	0.057s
12	4	50%, 20%, 30%	[1ms, 10ms, 1000ms]	3.981s	3.981s	6.227s
13	5	5%, 50%, 45%	[1ms, 5ms, 5ms]	0.074s	0.038s	0.052s
14	5	5%, 50%, 45%	[1ms, 10ms, 1000ms]	4.856s	4.785s	4.580s
15	5	25%, 30%, 45%	[1ms, 5ms, 5ms]	0.074s	0.048s	0.062s
16	5	25%, 30%, 45%	[1ms, 10ms, 1000ms]	4.831s	4.807s	7.442s
17	5	50%, 20%, 30%	[1ms, 5ms, 5ms]	0.037s	0.047s	0.087s
18	5	50%, 20%, 30%	[1ms, 10ms, 1000ms]	2.094s	2.073s	10.08s

it dictate the behavior of a large number of equivalent corresponding subgraphs. Each of the chain traversals incurs from 20 to 25 discrete communication events (starting from the end user request instantiating to the reception of the response) on average. Different sessions of simulation will generate different numbers of total sequences, based on user-defined service chain lengths.

3. The Terminal partition triggers each of the service chains by invoking the first microservice in the chain. Each time the vector is passed to a microservice residing in another segment, the associated "transport delay" is added to the delay value.
4. At the end of the simulation, the average response time is calculated.

This approach could also be extended in a similar way to calculate the delay introduced by software activities.

Table 1 shows a set of results with three different service chain lengths. Three distributions of segments are presented and they are organized in increasing capability of microservices placement of the Edge Segment. It move from 5% to 50% allocation at the Edge. Table 1 lists two sets of transport delay values between Terminal-to-Edge, Edge-to-Core, and Core-to-Cloud. One is extremely favorable to Edge deployment [1ms, 10ms, 1000ms] and others more favorable to utilizing the Cloud capabilities [1ms, 5ms, 5ms]. These options cover a broad set of application deployment cases: different approaches in packaging microservices (from service chains of length 3 to 5); increasing availability of Edge and Core resources; different transport delay values between the partitions; and different allocation of APIGTWs. Edge, Core and Cloud gateways (GTWs) are the API Gateway under control of SPs for placement of application coordination point (APIGTW). During the simulation, transport delay is accounted for user application request packet routing from one GTW to other i.e., Edge GTW to Core GTW.

The presented scenarios described in terms of service chain lengths, distribution of capacity between Edge, Core, Cloud, and different values of delay between the segments are only a part of the entire simulation. Some additional insights can be derived from the results in Table 1. The effectiveness of the usage of Edge resources for reducing the response time strictly depends on the size of the Edge infrastructure. The more resources are available at the Edge, the more the possibility that entire service chains could reside at the Edge. This favors the response time. On the other side, when the resources are available at the Edge and the Core, the response time is better. In many situations, the aggregation at the Core (i.e., index 1, 2, 3, 7, 9, 14, 15, 16 in Table 1) exhibit suitable because this segment can mediate between the delay for reaching the Cloud microservices. It also exploits a relatively small delay towards the end users. The Edge - Core - Cloud segmentation is seen as Regional - National - International segmentation. National data centers can benefit from relatively small delay and large capacity.

The results of Table 1 are also dependent on the heuristically adopted delays between the segments in the simulation process. When the delay is not larger in the case of APIGTW being placed in the Cloud, then the aggregation of microservices in the Cloud (especially with the scarcity of resources at the Edge and Core) makes sense. Following it, indexes 9, 13, 14, and 15 in Table 1 are insightful and suggest full aggregation of microservices in the Cloud segment, keeping the Cloud capabilities in mind. In intermediate conditions, the Core capability of mediating between the availability of resources and a better delay (compared to the Cloud) can play a relevant role.

In general terms, a better response time depends on the percentage of resources allocated to the different segments and the delay between them. If there are limited resources at the Edge, it is convenient to aggregate service chains in the Cloud even if the delay is much larger from Edge to Cloud. In the case of relevant capabilities at the Edge and Core, and a large delay to arrive at the Cloud segment, the allocation of service chains at the Edge and Core is much more convenient. The Core is also capable of compensating for the scarcity of resources at the Edge and limiting the delay in response time. If the Core is capable enough, then a good strategy is to allocate service chains including their APIGTW in this segment.

8 Possible Microservices Deployment Strategies

The simulation results have been instrumental for us in understanding the mechanisms and the issues related to a large chain of microservices and the effects of their placement on several segments with respect to the response time of the application. In order to further progress, there is a need to figure out simple rules that can help in choosing deployment options. Two approaches are presented in this section: the first one is based on the calculation of the response time of an application. It considers the length of the service chain, its possible deployment options to calculate by means of a cost function, and the response time for viable placements. The second one is based on the calculation of the application graph (in terms of relations and interactions of the components) and

then uses an affinity model to determine the components/microservices that should be collocated in order to reduce the response time.

8.1 Cost_Aware Service Chain Acceptance

The cost function method is based on the possibility of identifying the different modes of deploying a service chain of a given length on the identified segments. The service chain is represented as a "chain vector" where each microservice is substituted by the indication of where it may be placed. A vector of this form [0, 3, 2, 3, 1] represents a service chain whose starting element(index = 0) is on the Terminal Segment, the first element of the service chain (index = 1) is placed in the Edge (represented by the value 3), the second element (index = 2) is placed in the Core Segment (index = 2), the third one (index = 3) is placed in the Cloud, and the last microservice (index = 4) is in the Edge (represented by value 1). With this mechanism, any service chain can be mapped to a possible placement. Given a service chain of a specific length, it is possible to calculate all the possible placement options.

A Delay Matrix is used in Figure 8 for the computation of a cost function. It is created based on the expected delays between the segments. The element identified by the $[i, j]$ matrix indexes represents the estimated delay for a microservice deployed in segment i to interact with a microservice deployed in segment j. This matrix can represent the average delays experienced by the different segments. Besides the Delay Matrix, a cost vector C can be introduced. It represents the costs associated with the hosting of components in one of the different segments. In a system with the Terminal Segment and the Edge, Core, and Cloud ones, a vector of length 4 can be used. The first slot represents the cost of placing a component in the Terminal Segment, the second slot represents the Edge Segment's costs, and so on. The users are requested to provide the number of components of the service chain to be deployed on the system, the maximum acceptable value of the response time, and an acceptable measure of Acceptable Costs. To find the APIGTW response time and associated costs, the procedure follows the following steps:

1. All the possible sequences of k elements (length of the service chain) over n segments (in this case three segments: Edge, Core and Cloud) are calculated.
2. Each chain is serialized in a single-dimension "chain vector" and a zero column is inserted in the first place (the service chain will be triggered by the Terminal Segment).

$$\begin{aligned} \text{DelayAPIGTW} = 2\ (&D[chainvector[0]] \\ &[chainvector[1]] + \sum_{i=2}^{\text{chainLength}} \\ &D[chainvector[i]][chainvector[i+1]]) \end{aligned} \tag{1}$$

$$\text{CostAPIGTW} = \sum_{i=1}^{\text{chainLength}} C[chainvector[i]] \tag{2}$$

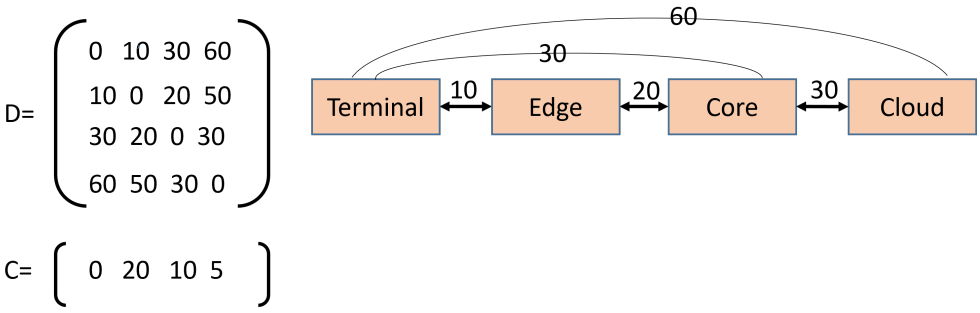

Fig. 8 Delay and cost matrix

3. The chain vector is fed to equation 1 and 2 which compute the delay and cost associated with the placement of the sequence in segments.
4. The calculated response time and costs from the iterative computation are compared with user-defined values to determine the acceptability of the specific chain vector.

A decision on the best deployment configuration could be made in terms of lower response time and acceptable costs. A cost-based decision will select placement options that have lower costs (in the cost Matrix C, the lower costs are associated with placements in the Cloud). Different evaluations of these parameters can be made in order to select the less expensive or the most performing placement or any balanced decision.

The cost matrix is a possible criterion in this procedure. The nature of the cost has been associated with the deployment cost in a segment in this simple formulation under the assumption that a deployment on the Edge may be more expensive than a deployment on Core that is more expensive than a placement on the Cloud. However, the cost function(s) can depend on different KPIs (e.g., carbon footprint of the segment, power consumption, and others) and the SP wants to consider and make them part of its offering to customers during the evaluation of chains.

8.2 Affinity Based Experiment in Kubernetes Cluster

With the previous analysis, a set of insights on how to take care of the response time for componentized applications has been derived from simulation and calculation of costs. There is a need to check what approach to take in real implementation. A distributed application executed on a cluster of different machines, envisioning the work carried out in [61].

The Sock-Shop application[2] (an e-commerce application) has been chosen to observe the microservices interactions, with a focus on the behavior of an APIGTW. According to [67], Sock-Shop is one of the most suitable reference applications for research in existing microservice placement solutions and testing novel Cloud-native solutions. It is composed of 13 distinct microservices (i.e., user, front-end, shipping, catalog, order, databases, and auxiliary microservices) developed in several languages (Java, Go, and Node.js). These microservices exchange parameters through HTTP

[2] https://github.com/microservices-demo/microservices-demo

Table 2 System used in the deployment of Sock-Shop

Server	Operating System	CPU	RAM
Dell Precision 5820	Ubuntu 22 LTS	16	130 GB
ProLiant MicroServer Gen10	Ubuntu 22 LTS	8	12GB

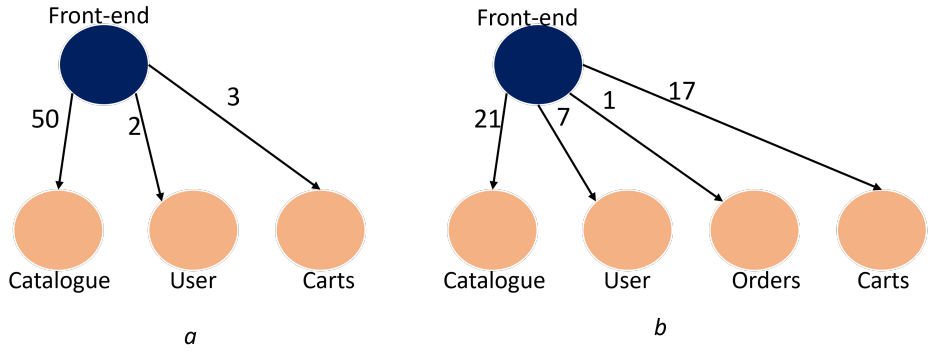

Fig. 9 A microservices call graph illustration of sock-shop application

and REST APIs. In our experimentations, the Sock-Shop has been deployed in a Kubernetes-based (K3s) cluster. The lab cluster comprises two servers and a set of smaller machines. The K3s orchestrator is capable of allocating different microservices according to specific configurations defined by the YAML language. The resource capabilities of the used servers are shortly depicted in Table 2.

The Istio service mesh has been used to collect data about the interaction of each microservice with the others. Istio integrates monitoring capabilities, a sort of proxy (Envoy[68]) in each microservice. It controls the communication between microservices and is capable of aggregating the microservice telemetry data. An Istio-enabled application, Jaeger [3] is used to represent the application as a graph and to report the number of calls for microservices in the graph.

To interact with the deployed microservices and compute an interaction graph, Load_Test[4] is used. It simulates the concurrent user's requests for the applications. HTTP traffic has been emulated in order to analyze the behavior of the application with requests from different numbers of users. One of the samples has considered traffic generated by 50 clients, each placing 7 requests for the front-end microservice. The value 7 determines the number of requests to run before terminating the tests. This configuration was instrumental in collecting a number of calls for each microservice's API, as presented in Figure 9.*a*. Secondly, we also consider user's typical interaction with an e-commerce website. A user interacts with the Sock-Shop application for 2 minutes, and consequently, various microservices are called (Figure 9.*b*). This configuration has offered the possibility of creating a graph of the Sock-Shop application. The graph clearly shows the relationship between different microservices and the invocations between different functionalities. The graph is a representation of the structure of the service chain. The number of calls from the Frontend microservice to the Catalogue

[3] https://github.com/jaegertracing/jaeger
[4] https://github.com/microservices-demo/load-test

Table 3 Average response time of collocated and distributed applications

Users	Per user HTTP requests	Collocated FrontEnd delay(ms)	Collocated Catalogue delay(ms)	Distributed FrontEnd delay(ms)	Distributed Catalogue delay(ms)
10	100	46.75	86.73	74.07	171.32
20	100	78.41	145.69	136.01	339.76
30	100	100.75	195.02	191.57	443.78
40	100	133.94	242.14	260.07	549.48
50	100	162.51	295.32	307.23	690.02

microservice is the most frequent of all the monitored occurrences (using Load_Test or directly interacting with the application by means of a client application). Considering the relevant number of invocations, as depicted in Figure 9.*a*, the placement of these two components can highly affect the response time of the entire application. For this reason, two experiments have been conducted: in the first one, the placement of the Frontend and Catalogue microservices was in the same segment (in this case, a specific server); in the second one, the deployment of these two most frequently invoked microservices was in two different servers. The average communication delay between the two used servers is 0.870ms (computed through ICMP protocol). The results of the experiments are represented in Table 3. We observed that with the increase in the number of users and interactions, the microservices' response time also started increasing gradually. Either in the case of default distribution of sock-shop through Kubernetes, or our deliberate collocation and distribution, with regard to response time, we found that placing Forntend and Catalogue microservices together in the same node is a winning strategy. We observed that with a larger number of users, the average response time of the catalogue service reaches 690.22ms per user if it is mapped on a server isolated from the Frontend.

This was the first step in the analysis of an affinity-based approach (number of calls in Figure 9). Additional experiments with the other microservices placed in different locations can provide a better understanding of the value of this approach.

-Initial elements for a strategy for improved placement of applications

The deployment of large distributed and componentized applications in an Edge-Cloud Continuum can be based on two integrated approaches: the knowledge of the system organization (e.g., the size of the Edge and Core, the availability of slots in these segments, and the probability that entire service chains can be deployed closer to the user) and the development, during the testing phase of the application, of a complete application graph. These two elements could be considered as complementary constraints for optimizing the placement of an application. The developers can consider the number of invocations between microservices during the application development and testing phases. The statistics (i.e., frequency) of invoked functions, and the emulation of expected traffic can help the developers to better understand the behavior of the applications and define a set of constraints in terms of deployment. The SPs could use these indications (and possibly the application graph) to optimize the deployment from the application perspective. The SP can foresee the interacting behavior of microservices and decide their collocation in the most convenient segments. This deployment approach can significantly represent the mapping of applications to the available resources. There are a few existing works that only focus on providing the

microservices description, for instance, [8] specify the response time in the application template.

9 Insights On Coordinated Microservices Placement

Studying the behavior of applications shows that the response time of an application depends on how the service chain is organized and deployed. The length of the service chain as well as the capacity of the Edge and Core segments are important variables to consider in the placement algorithms. Two major patterns of interaction for the applications have been considered: chaining and controlled execution (the APIGTW). These are the most used ones in the deployment of microservices and the results of the simulation have a wide application. An important finding is related to the placement of APIGTWs. In this case, the co-location of the Frontend functions and the most invoked components is highly valuable and recommended. However, the approach should also consider the topology of the infrastructure and the possible need to replicate the placement of APIGTWs. The replicated APIGTWs placement increases the efforts of SPs to control the operations of these important points of contact with clients. In the case of deployment of APIGTWs, their placement in other partitions could be constrained and dictated by other requirements.

9.1 Insights on microservice allocation

The simulation tool has been extensively used to investigate the different patterns or diverging behaviors, depending on the variation of the different parameters. The simulation yields the following conclusions:.

9.1.1 Slot distribution scenarios

The more slots at the Edge, the lower the transport delay. This statement is generally true, but with the understanding that the distribution of microservices has a huge impact on the total response time. When microservices are widely distributed in different segments (the case of long service chains), the delay can rapidly grow and be higher than less distributed placements in farther away segments. In general, the aggregation of the entire service chain into a single segment is a viable option. This holds true also for the all in the Cloud option, which can be a very competitive solution when Edge resources are scarce and the service chain is long.

9.1.2 Low Delay Scenario

The lower the delay at the Edge, the lower the total response time. This assumption generally holds true, but it strongly depends on the delay in reaching other segments and the level of distribution of the microservice chains. If the difference between Edge and Cloud delay is very large, then placing microservices closer to the Edge is a good option. However, the distribution of microservices on different segments has an increasingly higher influence on the response time; e.g., the invocation of a single microservice residing on a Cloud node with a huge delay can highly affect the total response time, making the deployment of all other microservices at the Edge irrelevant

and expensive. For scenarios in which the difference in delay is not so large (e.g., the 1-5-5 ms scenario), the rule of thumb of aggregating the service chains into a single segment (even with higher delay) seems to be a viable solution.

9.1.3 Length of Service Chain

The longer the chain, the higher the delay. This rule holds for distributed service chains spanning over more segments. Under the assumptions of zero execution and system delay, if microservices are interacting locally, the aggregation of microservices into a single system is a winning strategy. The second step in the strategy is to deploy the remaining microservices of the service chain in the adjacent segment with the lowest transport delay.

The creation and usage of a graph representing the application and its invocation load is another important aspect that can greatly contribute to the improvement of applications' response time. This approach should be implemented and realized during the development and testing phase. The application graph could be an input to the orchestrators to optimize the placement of the application in the distributed system. Some tools are currently used more for monitoring application behavior than for detecting optimized placement.

10 Conclusion

This paper was a deep dive into the microservices placement problem in the context of the Edge-Cloud Continuum and more in general in different network segments. The study confirmed that the availability of computing capabilities in more than one computing segment enables flexible microservice deployment strategies; however, the identification of an initial efficient placement of microservices is challenging. We have contextualized the problem in two different options: microservice service chains and coordinated microservices (using an APIGTW). In both cases, the response time was mainly determined by the number of interactions between microservices. If the deployments are greater than two and distributed in more than one segment, then it is better to move the microservices placed on the Edge towards the Core or the Cloud. This will improve the response time and/or free up valuable resources at the Edge. The analysis of computed response times using simulation and real-time deployment scenarios recommends placing the APIGTW close to microservices that pertain to the higher affinity. Closer to the Edge, the more valuable resources are needed and, most likely, the microservices and the APIGTW itself need to be replicated to avoid continuous access to Core or Cloud functions from the Edge. A good response time requires a high aggregation of microservices towards the Edge. The extensive simulations show that placing APIGTWs in segments with a lower response time is an advantage. However, for large applications (with several microservices), the number of interactions between functions placed on different segments may increase, leading to the need to aggregate the components. In this case, the aggregation at the Cloud level could result in convenience for the optimized response time, as well as for a higher level of centralization and management of the replications and privacy and control of important data. The

simulation results, in combination with microservice affinity computation, and criteria posed for acceptability of application response time, can help the SPs choose an acceptable microservices chain placement.

11 Future Work

In the extension of this work, additional analysis of types of microservices (chains and controller type) will be carried out to identify the different types of control and interaction. Two types of reusable functions (and related microservices) will be considered. First, the static ones, i.e., those that are directly controlled and instantiated by the SP according to its policies and constraints. Secondly, the dynamic ones, i.e., those that can be instantiated within a service chain under the control of the application developers. The former ones do constrain the service chains to a specific configuration. A second step of the study will be to integrate Artificial Intelligence (AI) techniques to predict the behavior of service chains and to optimize the deployment configuration. Due to the agility and fast integration of microservices, each service chain will compete for scarce resources (the assumption is that Edge resources are scarcer than Cloud resources). The individual application optimization policies will be aligned with global policies for optimizing the allocation, deployment, or migration of microservices. These experiments, simulations, and deployments will be tested within the experimental lab, which is a part of the Edge-Cloud infrastructure.

Acknowledgements. This research work is supported by Project CLOUD CONTINUUM SOUVERAIN ET JUMEAUX NUMRIQUES under Grant AMI CLOUD-1 C2JN (DOS0179613/00,DOS0179612/00), and the DOCTE6G. We are thankful to the C2JN project partners for consolidating the computing continuum to deploy and validate the yield of this research work.

Author Contribution. Syed Mohsan Raza: Conceptualization, Methodology, Visualization, Writing-original draft. Roberto Minerva: Conceptualization, Writing-original draft, Validation, Visualization. Barbara Martini:Conceptualization, Editing, Writing-original draft, Investigation, Validation. Noel Crespi: Methodology, Supervision, Investigation, Validation.

Declaration of interests. The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Balouek-Thomert, D., Renart, E.G., Zamani, A.R., Simonet, A., Parashar, M.: Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. The International Journal of High Performance Computing Applications **33**(6), 1159–1174 (2019)

[2] Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, pp. 13–16 (2012)

[3] Netaji, V.K., Bhole, G.P.: A comprehensive survey on container resource allocation approaches in cloud computing: State-of-the-art and research challenges. In: Web Intelligence, vol. 19, pp. 295–316 (2021). IOS Press

[4] Di Francesco, P., Lago, P., Malavolta, I.: Migrating towards microservice architectures: an industrial survey. In: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 29–2909 (2018). IEEE

[5] Kaur, K., Guillemin, F., Rodriguez, V.Q., Sailhan, F.: Latency and network aware placement for cloud-native 5g/6g services. In: 2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC), pp. 114–119 (2022). IEEE

[6] Aksakalli, I.K., Çelik, T., Can, A.B., Tekinerdoğan, B.: Deployment and communication patterns in microservice architectures: A systematic literature review. Journal of Systems and Software **180**, 111014 (2021)

[7] Fu, Y., Shan, Y., Zhu, Q., Hung, K., Wu, Y., Quek, T.Q.: A distributed microservice-aware paradigm for 6g: Challenges, principles, and research opportunities. IEEE Network (2023)

[8] Bulej, L., Bureš, T., Filandr, A., Hnětynka, P., Hnětynková, I., Pacovskỳ, J., Sandor, G., Gerostathopoulos, I.: Managing latency in edge–cloud environment. Journal of systems and software **172**, 110872 (2021)

[9] Alvarado-Valiente, J., Romero-Álvarez, J., Moguel, E., García-Alonso, J., Murillo, J.M.: Technological diversity of quantum computing providers: a comparative study and a proposal for api gateway integration. Software Quality Journal, 1–21 (2023)

[10] Pallewatta, S., Kostakos, V., Buyya, R.: Microfog: A framework for scalable placement of microservices-based iot applications in federated fog environments. Journal of Systems and Software **209**, 111910 (2024)

[11] Laso, S., Flores, D., Garcia-Alonso, J., Murillo, J.M., Berrocal, J.: Deploying apis: Edge vs cloud environments. MMTC Communications-Frontiers **19** (2019)

[12] Cheng, K., Zhang, S., Liu, M., Gu, Y., Wei, L., Cheng, H., Liu, K., Song, Y., Shi, X., Zhu, A., *et al.*: Geoscale: Microservice autoscaling with cost budget in geo-distributed edge clouds. IEEE Transactions on Parallel and Distributed Systems **35**(4), 646–662 (2024)

[13] Peng, K., Wang, L., He, J., Cai, C., Hu, M.: Joint optimization of service deployment and request routing for microservices in mobile edge computing. IEEE

Transactions on Services Computing (2024)

[14] Wang, Y., Shu, Z., Chen, S., Lin, J., Zhang, Z.: A cost and demand sensitive adjustment algorithm for service function chain in data center network. Computer Networks **242**, 110254 (2024)

[15] Brogi, A., Forti, S., Ibrahim, A.: Optimising qos-assurance, resource usage and cost of fog application deployments. In: Cloud Computing and Services Science: 8th International Conference, CLOSER 2018, Funchal, Madeira, Portugal, March 19-21, 2018, Revised Selected Papers 8, pp. 168–189 (2019). Springer

[16] Brondolin, R., Santambrogio, M.D.: Presto: a latency-aware power-capping orchestrator for cloud-native microservices. In: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), pp. 11–20 (2020). IEEE

[17] Nassereldine, A., Diab, S., Baydoun, M., Leach, K., Alt, M., Milojicic, D., El Hajj, I.: Predicting the performance-cost trade-off of applications across multiple systems. In: 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 216–228 (2023). IEEE

[18] Gong, Y., Bian, K., Hao, F., Sun, Y., Wu, Y.: Dependent tasks offloading in mobile edge computing: a multi-objective evolutionary optimization strategy. Future Generation Computer Systems **148**, 314–325 (2023)

[19] Souza, P.S., Ferreto, T., Calheiros, R.N.: Edgesimpy: Python-based modeling and simulation of edge computing resource management policies. Future Generation Computer Systems (2023)

[20] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., Pallickara, S.: Serverless computing: An investigation of factors influencing microservice performance. In: 2018 IEEE International Conference on Cloud Engineering (IC2E), pp. 159–169 (2018). IEEE

[21] Roman, D., Song, H., Loupos, K., Krousarlis, T., Soylu, A., Skarmeta, A.F.: The computing fleet: Managing microservices-based applications on the computing continuum. In: 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C), pp. 40–44 (2022). IEEE

[22] Nath, S.B., Chattopadhyay, S., Karmakar, R., Addya, S.K., Chakraborty, S., Ghosh, S.K.: Ptc: Pick-test-choose to place containerized micro-services in iot. In: 2019 IEEE Global Communications Conference (GLOBECOM), pp. 1–6 (2019). IEEE

[23] Pallewatta, S., Kostakos, V., Buyya, R.: Qos-aware placement of microservices-based iot applications in fog computing environments. Future Generation Computer Systems **131**, 121–136 (2022)

[24] Canali, C., Di Modica, G., Lancellotti, R., Rossi, S., Scotece, D.: A validated performance model for micro-services placement in fog systems. SN Computer Science **4**(4), 417 (2023)

[25] Salaht, F.A., Desprez, F., Lebre, A.: An overview of service placement problem in fog and edge computing. ACM Computing Surveys (CSUR) **53**(3), 1–35 (2020)

[26] Niu, Y., Liu, F., Li, Z.: Load balancing across microservices. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications, pp. 198–206 (2018). IEEE

[27] Islam, M.M., Ramezani, F., Lu, H.Y., Naderpour, M.: Optimal placement of applications in the fog environment: A systematic literature review. Journal of Parallel and Distributed Computing **174**, 46–69 (2023)

[28] Villari, M., Celesti, A., Tricomi, G., Galletta, A., Fazio, M.: Deployment orchestration of microservices with geographical constraints for edge computing. In: 2017 IEEE Symposium on Computers and Communications (ISCC), pp. 633–638 (2017). IEEE

[29] Khan, M.G., Taheri, J., Al-Dulaimy, A., Kassler, A.: Perfsim: A performance simulator for cloud native microservice chains. IEEE Transactions on Cloud Computing **11**(2), 1395–1413 (2021)

[30] Sampaio, A.R., Rubin, J., Beschastnikh, I., Rosa, N.S.: Improving microservice-based applications with runtime placement adaptation. Journal of Internet Services and Applications **10**(1), 1–30 (2019)

[31] Marchese, A., Tomarchio, O.: Network-aware container placement in cloud-edge kubernetes clusters. In: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 859–865 (2022). IEEE

[32] Marchese, A., Tomarchio, O.: Application and infrastructure-aware orchestration in the cloud-to-edge continuum. In: 2023 IEEE 16th International Conference on Cloud Computing (CLOUD), pp. 262–271 (2023). IEEE

[33] Ding, Z., Wang, S., Jiang, C.: Kubernetes-oriented microservice placement with dynamic resource allocation. IEEE Transactions on Cloud Computing (2022)

[34] Bufalino, J., Di Francesco, M., Aura, T.: Analyzing microservice connectivity with kubesonde. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 2038–2043 (2023)

[35] Rossi, F., Cardellini, V., Presti, F.L., Nardelli, M.: Geo-distributed efficient deployment of containers with kubernetes. Computer Communications **159**, 161–174 (2020)

[36] Chowdhury, S.R., Salahuddin, M.A., Limam, N., Boutaba, R.: Re-architecting nfv ecosystem with microservices: State of the art and research challenges. IEEE Network **33**(3), 168–176 (2019)

[37] Sheoran, A., Sharma, P., Fahmy, S., Saxena, V.: Contain-ed: An nfv micro-service system for containing e2e latency. ACM SIGCOMM Computer Communication Review **47**(5), 54–60 (2017)

[38] Kaur, K., Guillemin, F., Sailhan, F.: Dynamic migration of microservices for end-to-end latency control in 5g/6g networks. Journal of Network and Systems Management **31**(4), 84 (2023)

[39] Bhamare, D., Samaka, M., Erbad, A., Jain, R., Gupta, L., Chan, H.A.: Optimal virtual network function placement in multi-cloud service function chaining architecture. Computer Communications **102**, 1–16 (2017)

[40] Zuo, X., Su, Y., Wang, Q., Xie, Y.: An api gateway design strategy optimized for persistence and coupling. Advances in Engineering Software **148**, 102878 (2020)

[41] Tomić, M., Dimitrieski, V., Vještica, M., Župunski, R., Jeremić, A., Kaufmann, H.: Towards Applying API Gateway to support Microservice Architectures for Embedded Systems. ICIST (2022)

[42] Xu, R., Jin, W., Kim, D.: Microservice security agent based on api gateway in edge computing. Sensors **19**(22), 4905 (2019)

[43] Zhao, J., Jing, S., Jiang, L.: Management of api gateway based on micro-service architecture. In: Journal of Physics: Conference Series, vol. 1087, p. 032032 (2018). IOP Publishing

[44] Moreira, P., Ribeiro, A., Silva, J.M.: Age: Automatic performance evaluation of api gateways. In: 2023 IEEE Symposium on Computers and Communications (ISCC), pp. 405–410 (2023). IEEE

[45] Pallewatta, S., Kostakos, V., Buyya, R.: Placement of microservices-based iot applications in fog computing: A taxonomy and future directions. ACM Computing Surveys **55**(14s), 1–43 (2023)

[46] Doan, T.V., Bajpai, V., Crawford, S.: A longitudinal view of netflix: content delivery over ipv6 and content cache deployments. In: IEEE INFOCOM 2020-IEEE Conference on Computer Communications, pp. 1073–1082 (2020). IEEE

[47] (ISG), N.F.V.N.E.I.S.G.: Management and Orchestration; Architectural Framework Specification @ONLINE. https://www.etsi.org/deliver/etsi_gs/NFV/001_099/006/03.06.01_60/gs_nfv006v030601p.pdf

[48] Paganelli, F., Ulema, M., Martini, B.: Context-aware service composition and

delivery in ngsons over sdn. IEEE Communications Magazine **52**(8), 97–105 (2014)

[49] Hiren Surti, Pack Janes, Tom Craft, Tom Widawsky: Types and locations of edge data centers. Technical report, Telecommunications Industry Association, TIA (October 2019)

[50] Santoyo-González, A., Cervelló-Pastor, C.: Edge nodes infrastructure placement parameters for 5g networks. In: 2018 IEEE Conference on Standards for Communications and Networking (CSCN), pp. 1–6 (2018). IEEE

[51] Isazadeh, A., Ziviani, D., Claridge, D.E.: Global trends, performance metrics, and energy reduction measures in datacom facilities. Renewable and Sustainable Energy Reviews **174**, 113149 (2023)

[52] Gharbaoui, M., Martini, B., Cecchetti, G., Castoldi, P.: Resource orchestration strategies with retrials for latency-sensitive network slicing over distributed telco clouds. IEEE Access **9**, 132801–132817 (2021)

[53] Bilal, K., Khalid, O., Erbad, A., Khan, S.U.: Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers. Computer Networks **130**, 94–120 (2018)

[54] Plauth, M., Feinbube, L., Polze, A.: A performance survey of lightweight virtualization techniques. In: European Conference on Service-Oriented and Cloud Computing, pp. 34–48 (2017). Springer

[55] Arora, S., Ksentini, A., Bonnet, C.: Cloud native lightweight slice orchestration (cliso) framework. Computer Communications (2023)

[56] Shadija, D., Rezai, M., Hill, R.: Microservices: granularity vs. performance. In: Companion Proceedings of The10th International Conference on Utility and Cloud Computing, pp. 215–220 (2017)

[57] López, P.G., Sánchez-Artigas, M., París, G., Pons, D.B., Ollobarren, Á.R., Pinto, D.A.: Comparison of faas orchestration systems. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pp. 148–153 (2018). IEEE

[58] Liu, D.H., Levy, A., Noghabi, S., Burckhardt, S.: Doing more with less: Orchestrating serverless applications without an orchestrator. In: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pp. 1505–1519 (2023)

[59] Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S.: Automated functional and robustness testing of microservice architectures. Journal of Systems and Software **207**, 111857 (2024)

[60] Luo, S., Xu, H., Lu, C., Ye, K., Xu, G., Zhang, L., He, J., Xu, C.: An in-depth study of microservice call graph and runtime performance. IEEE Transactions on Parallel and Distributed Systems **33**(12), 3901–3914 (2022)

[61] Colarusso, C., De Caro, A., Falco, I., Goglia, L., Zimeo, E.: A distributed tracing pipeline for improving locality awareness of microservices applications. Software: Practice and Experience (2024)

[62] Montesi, F., Weber, J.: Circuit breakers, discovery, and api gateways in microservices. arXiv preprint arXiv:1609.05830 (2016)

[63] Adib, D.: How Does Edge Computing Architecture Impact Latency. https://stlpartners.com/articles/edge-computing/how-does-edge-computing-architecture-impact-latency/

[64] Sanchez-Gomez, J., Marin-Perez, R., Sanchez-Iborra, R., Zamora, M.A.: Mec-based architecture for interoperable and trustworthy internet of moving things. Digital Communications and Networks **9**(1), 270–279 (2023)

[65] Gan, Y., Delimitrou, C.: The architectural implications of cloud microservices. IEEE Computer Architecture Letters **17**(2), 155–158 (2018)

[66] Varga, A.: A practical introduction to the omnet++ simulation framework. In: Recent Advances in Network Simulation, pp. 3–51. Springer, ??? (2019)

[67] Aderaldo, C.M., Mendonça, N.C., Pahl, C., Jamshidi, P.: Benchmark requirements for microservices architecture research. In: 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), pp. 8–13 (2017). IEEE

[68] Merino, X., Otero, C., Nieves-Acaron, D., Luchterhand, B.: Towards orchestration in the cloud-fog continuum. In: SoutheastCon 2021, pp. 1–8 (2021). IEEE