KubeTwin: A Digital Twin framework for Kubernetes deployments at scale

Davide Borsatti, *Member, IEEE*, Walter Cerroni, *Senior Member, IEEE*, Luca Foschini, *Senior Member, IEEE*, Genady Ya. Grabarnik, *Senior Member, IEEE*, Lorenzo Manca, Filippo Poltronieri, *Member, IEEE*, Domenico Scotece, *Member, IEEE*, Larisa Shwartz, *Senior Member, IEEE*, Cesare Stefanelli, *Member, IEEE*, Mauro Tortonesi, *Member, IEEE*, Mattia Zaccarini, *Student Member, IEEE*

***Abstract*—Kubernetes is a well-known orchestration and management solution for complex and large-scale service architectures in the Cloud Continuum. While it provides very valuable functions from the operation perspective, the high number of control loops it implements significantly enlarges the already wide space of configuration parameters and policies to consider for management purposes. We argue that optimizing complex Kubernetes deployments considering a multi-cloud and edge computing environment would significantly benefit from a Digital Twin approach, enabling an accurate virtual representation of a Kubernetes application to optimize its deployment and management policies. Towards that goal, this work illustrates the design of KubeTwin, a framework to implement Digital Twins of Kubernetes deployments. Furthermore, we present a validation of KubeTwin in a Multi-access Edge Computing (MEC) scenario, which shows its soundness in reenacting realistic Digital Twins of complex and highly distributed Kubernetes deployments. We believe that KubeTwin can provide useful guidance to the research community working in this field.**

***Index Terms*—Digital Twin, Service Management and Orchestration, Multi-access Edge Computing, Kubernetes, Simulation.**

I. Introduction

In the ongoing and ever-accelerating process of network softwarization, researchers are turning their attention towards *Digital Twins* [1]. The Digital Twin concept has emerged in industry 4.0 as a high-accuracy virtual representation and software companion for physical assets related to applications ranging from maintenance [2] to process and network optimization [3]–[5]. However, more recently, the term has also been applied to software platforms and digital assets. Along that way, some authors have already proposed Digital Twins for mimicking and configuring/interacting with either networks, applications, or both [6]. In particular, Digital Twins can address a wide range of situations by leveraging what-if scenario analysis and employing different mechanisms, spanning from performance optimization to chaos engineering [7]–[9].

D. Borsatti, W. Cerroni, L. Foschini, L. Manca, and D. Scotece are with the University of Bologna, Bologna, Italy e-mail: {davide.borsatti, walter.cerroni, luca.foschini, lorenzo.manca, domenico.scotece}@unibo.it.

G. Ya. Grabarnik is with St. John's University, Queens, NY, USA e-mail: grabarng@stjohns.edu.

F. Poltronieri, C. Stefanelli, M. Tortonesi, and M. Zaccarini are with the University of Ferrara, Ferrara, Italy e-mail: {filippo.poltronieri, cesare.stefanelli, mauro.tortonesi, mattia.zaccarini}@unife.it.

L. Shwartz is with IBM TJ Watson Research Center, NY, USA e-mail: lshwart@us.ibm.com.

Within this research avenue, a relatively recent development is to consider large-scale applications. Nowadays, large-scale applications are deployed on complex hybrid Cloud scenarios, with many different public and private Cloud environments and dynamic workloads, that present several challenges from the perspective of identifying optimal deployment configurations [7]. In practice, it is very hard to find out the optimal configuration of computational and network resources for a large-scale application before their actual deployment.

This task is even more complicated by the adoption of sophisticated orchestration platforms, such as Kubernetes. Kubernetes is becoming the de-facto solution for service management and orchestration, and it is increasingly proposed as a platform for a wide range of applications: from NFV implementation [10] to the tactical edge domain [11]. While this presents several advantages from the service provider perspective, it makes it even more difficult to assess upfront the proper configuration of a large-scale deployment, because its accurate evaluation must consider aspects that go well beyond the provisioning of resources, such as the number of VMs to rent, their prices, etc., and needs to evaluate the runtime impact of Kubernetes's control loops on the application behavior.

To address the above challenges, we claim the need to design novel *Digital Twin solutions purposely implemented by considering Kubernetes and the requirements of Kubernetes-based applications*. Those solutions would allow us to accurately capture the state of an existing Kubernetes-based IT application deployment through a virtual object with a smaller footprint and be capable of running what-if scenario analysis much faster than on a physical testbed. This would allow the efficient and parallel evaluation of the behavior of the Digital Twin using different configuration parameters or even modified components, with many relevant applications ranging from design feedback to resource optimization and chaos engineering.

This paper presents KubeTwin, a comprehensive framework designed to implement Digital Twins of Kubernetes-based deployments. KubeTwin emulates the Kubernetes orchestration functions and networking behaviors to be used as a guideline for service providers to accurately evaluate the impact of complex and large-scale Kubernetes deployment scenarios. It allows the definition of applications through a declarative description, which is semantically equivalent to Kubernetes', to reenact and assess them through both generic and application-specific (and user-defined) Key Performance Indicators (KPIs).

KubeTwin enables its adopters to leverage the Digital Twin to identify optimized deployment configurations and evaluate different scheduling policies in complex computing scenarios where resources can be distributed at many levels, such as edge servers and cloud data centers.

The contributions of this paper are manifolds. Firstly, this manuscript describes the implementation of the KubeTwin framework. Secondly, we present a comprehensive use case that contains not only the description of a container-based application that can be used as a reference for testing [12] but also a detailed Compute Continuum scenario composed of edge and cloud resources.

We evaluated KubeTwin in a reference use case reenacting a distributed Multi-access Edge Computing (MEC) application deployment. MEC is an Industry Standardization Group (ISG) backed by ETSI and other industrial partners aiming to provide the capabilities of hosting IT services at the edge of the network [13], [14]. These applications' proximity to the network's access part offers reduced latency and increased bandwidth. Furthermore, MEC offers standardized APIs that applications can consume to get real-time radio access network information. The application deployment we considered includes a hybrid computing scenario with a three-tier MEC environment and two remote Cloud data centers, representing a relevant use case to test the capabilities of KubeTwin.

Our evaluation has shown that KubeTwin can replicate complex and large-scale IT infrastructures at both the application and orchestration platform levels. This allows for capturing key performance indicators (KPIs) that are essential to gaining an understanding of how the system operates. The soundness of the KubeTwin framework as a Digital Twin for Kubernetes deployments is thus demonstrated, and it presents interesting research opportunities for future investigation.

II. The Case for a Kubernetes Digital Twin

There is no clear-cut definition of the Digital Twin concept [15], and several ones have been proposed over the years (see Section 2 of [2] for a nicely written discussion). However, most of the definitions seem to agree that a Digital Twin is a system composed of 3 elements: a physical object (or system), a high-accuracy virtual representation of it (often obtained by adopting sophisticated simulation solutions), and an active bidirectional link between those elements, that allows aligning the state of the physical object and its virtual representation. Extra components for performance evaluation and optimization, or more in general decision-making, are often present. In literature, the term Digital Twin is, rather ambiguously, used to define the virtual representation element, the entire system, or both.

Let us note that in a Digital Twin, the link between the physical and virtual elements is bidirectional: it goes *from the real system to its virtual representation*, thus allowing the latter to keep track of state changes in the real system, and *from the virtual representation to the real system*, thus allowing to put in place changes in the system configuration that were explored in the digital twin and judged more satisfactory than the previous ones. It is important to note that while the link between the physical and virtual elements is bidirectional, the virtual element can be used to run disconnected or "offline" what-if scenario analyses. During this process, the virtual element explores several configurations without altering the physical element. Only when a satisfactory configuration is found will it be translated to the physical element.

In the network and service management domain, researchers and practitioners deal with large-scale and very complex networks and applications that exhibit quite a dynamic behavior, and they constantly explore new methodologies and tools that help them in the struggle to maintain and optimize their systems continuously [16]. For this reason, they are increasingly adopting sophisticated orchestration solutions such as Kubernetes, which provides functional management tools, including network management, automated deployment, and autoscaling assistance that simplify the management of complex and large-scale applications. Kubernetes adopts a declarative approach to application deployment, implementing a wide range of complex control loops that continuously monitor applications and modify their deployments to match the desired state. Kubernetes adopters are required to provide a detailed description of their application in terms of container specifications, deployment configurations, and expected KPIs. However, while Kubernetes provides valuable functions from the operation automation perspective, including dynamic behaviors such as autoscaling and software update management, the large number of control loops it implements significantly enlarges the already vast space of configuration parameters and policies to consider for management purposes.

We argue that this domain would significantly benefit from Digital Twin approaches, extending and redefining the concept as shown in Fig. 1. In this case, the physical system is itself a digital object: a large-scale Kubernetes-based deployment built on top of many microservices, which in turn are executed in a federation of Kubernetes clusters, typically with complex configurations that control orchestration and automation behavior at several levels: cluster, application, and software component. This represents a very intricate system with significant dynamic aspects due to varying workloads, complex service deployments, heterogeneous network fabric, etc., whose accurate reenactment requires capturing the system behavior at the application and platform levels. Creating a virtual representation of this system presents critical challenges at the simulation level (both from the performance and accuracy perspectives), as well as at the performance assessment and optimization/decision-making levels. In fact, optimization criteria are typically defined at the business level, thus resulting in an additional layer of complexity to the performance assessment and significantly complicating and enlarging the space of possible configurations to explore for decision-making.

Developing Digital Twins of Kubernetes-based deployments calls for innovative solutions capable of addressing the problems of accurately reenacting large-scale Kubernetes-based software deployment, evaluating their behavior under different operating conditions, identifying potentially more optimal configurations, and reifying the configuration changes by interacting with the control APIs of the physical Kubernetes cluster(s)

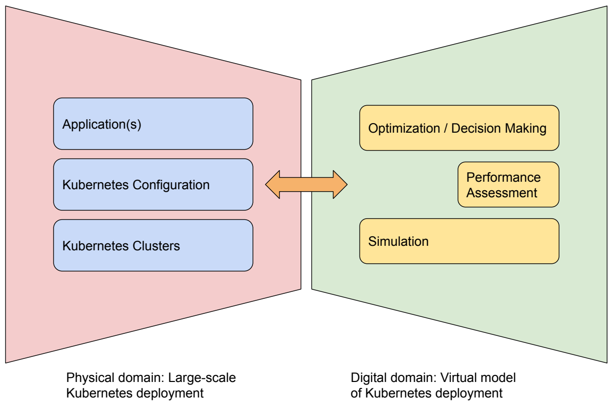
Application(s)
Kubernetes Configuration
Kubernetes Clusters
Optimization / Decision Making
Performance Assessment
Simulation
Physical domain: Large-scale Kubernetes deployment
Digital domain: Virtual model of Kubernetes deployment

Fig. 1. The Digital Twin concept redefined to consider Kubernetes-based software deployment.

considered. However, the effective development of Digital Twins requires sophisticated monitoring tools that can capture the behavior of the Kubernetes deployment, such as computing and network resource monitoring, performance estimation, and so on. The collected metrics will be used to define the Digital Twin and manage the Kubernetes deployment.

For this reason, we developed KubeTwin as a valuable framework to define and reenact Digital Twins of Kubernetes-based software deployment. KubeTwin is a comprehensive tool that can study the behavior of Kubernetes deployment using a Digital Twin approach to run what-if scenario analyses. This tool allows its adopters to identify optimized deployment configurations, evaluate various scheduling policies, experiment with different load-balancing algorithms, and finally translate these configurations and policies into Kubernetes specifications.

III. KubeTwin

KubeTwin is a framework that allows the creation of digital twins of Kubernetes-based software deployment and the evaluation of their performance in a deployment scenario of interest. To execute the digital twin, KubeTwin reenacts the behavior of a Kubernetes deployment at a very fine-grained level, simulating each service request as it travels. Particular attention was dedicated to the definition of Business Process Execution Language (BPEL)-like workflows on top of service components, to the implementation of control loops for service component replication, and the accurate modeling of network communication latency. This is to provide a higher level of accuracy when compared to the use of simpler regression models, which cannot simulate complex and dynamic scenarios, and to provide realistic insights regarding resource utilization and response times under changing conditions.

As Kubernetes, KubeTwin logically divides sets of computing nodes into *clusters*. Each cluster is identified by its name, type, location, and the number of computing nodes. In addition, a cluster defines a configurable amount of computing resources, e.g., CPU or GPU cores, to specify the performance of its computing nodes. We adopt this design choice to enable the modeling of different types of clusters that KubeTwin can use to deploy the application components. In particular, with KubeTwin, we can define two types of computing nodes: edge nodes, e.g., MEC servers hosted in small-size data centers close to the end-user premises, and medium- and large-size data centers located at Cloud facilities. This solution enables the reenacting of complex, heterogeneous, multi-cluster deployments. For example, it is conceivable that shortly a service provider could distribute an application by exploiting both MEC and cloud computing resources. In this case, nodes available at cloud facilities would likely provide much more computational resources than MEC nodes.

KubeTwin applications are defined using a declarative description, which is semantically equivalent to Kubernetes's, adopting a flexible approach that allows its users to specify different kinds of applications such as multi-tier web applications or complex money transfer management systems [7]. To this end, KubeTwin users will describe complex services as the composition of multiple microservices interacting together. In addition, KubeTwin groups user requests into workflows, which specify the coordinated execution of a subset of microservices.

Regarding the execution model, we envisioned that a digital twin created with KubeTwin could run in two operational modes: "offline" and "online". In the "offline" mode, KubeTwin functions independently from the live Kubernetes environment, utilizing historical data and predefined scenarios for conducting risk-free *what-if scenario analyses*, thus allowing service providers to explore potential outcomes and optimizations without impacting the actual deployment. We believe that the "offline mode" is a resource-efficient approach, ideal for training, planning, and testing changes in a simulated environment. On the other hand, when running in the "online" mode, KubeTwin operates in parallel with the live Kubernetes system, thus implementing a real-time bidirectional link between the physical and the virtual elements of the digital twin. In this mode, it continuously integrates and processes real-time data, enabling dynamic monitoring and fine-grained tuning of the Kubernetes deployment configurations. We are currently working on designing methods to model the microservices response time accurately to improve the accuracy of KubeTwin leveraging monitoring data from a Kubernetes environment. The first results can be found in our recent work [17] proposing an AI-based algorithm to estimate the response time of microservices that make up an application running on a Kubernetes cluster. By applying the response time distributions obtained by this algorithm to KubeTwin, we observed a significant improvement in the accuracy of the prediction made by the digital twin.

Let us point out that while both operational modes can bring valuable insights to optimize the behavior of a container-based application deployed on Kubernetes, the "offline" mode is the best choice for identifying the best configuration before deploying the application at scale.

KubeTwin is written in Ruby and distributed open-source (MIT license) on GitHub[1]. Ruby is a high-level, object-oriented programming language well known for its expres-

[1]https://github.com/DSG-UniFE/KubeTwin

siveness and ease of use. This makes it a good choice for developing complex systems like discrete event simulators. In fact, the dynamic nature of Ruby allows for code to be written in a more natural and readable way, reducing development time and improving code maintainability. We implemented KubeTwin as a single-process discrete-event simulator to reenact the behavior of a service managed on the top of the Kubernetes orchestration platform. KubeTwin users need to create a deployment file describing the service to provide, its configuration, the available clusters to allocate the service components, and the distribution of user requests.

A. Main Components

Figure 2 illustrates the interaction between KubeTwin components, which we designed to implement a realistic digital twin of the Kubernetes framework. We developed each element with the idea of making it as much compliant as possible with an actual implementation of Kubernetes. Specifically, KubeTwin implements these components to reenact the Kubernetes functionalities accurately, such as naming resolution, load balancing, scheduling, and so on.

First, a KTService is used to represent a Kubernetes Service. KubeTwin implements KTService as a list of associated pods that match a given selector, i.e., the KTService name. Each list element represents an “endpoint”, a symbolic link that maps a KTService with an associated pod. In addition, each KTService must define a load balancing policy to distribute the load of processing requests among associated pods. This load balancing policy could be one of the default ones implemented by Kubernetes or even user-defined rules that extend the default behavior, e.g., location-based load balancing.

Currently, KubeTwin implements a load balancing policy that mimics the default one implemented via iptables in Kubernetes. In addition, KubeTwin can also assign incoming requests to different pods in a round-robin fashion. Let us note that other policies can be easily defined using the KubeTwin framework. For instance, to minimize communication delay, KubeTwin users may want to specify a latency-based load-balancing policy that assigns incoming requests to pods running on computing nodes near the requesters.

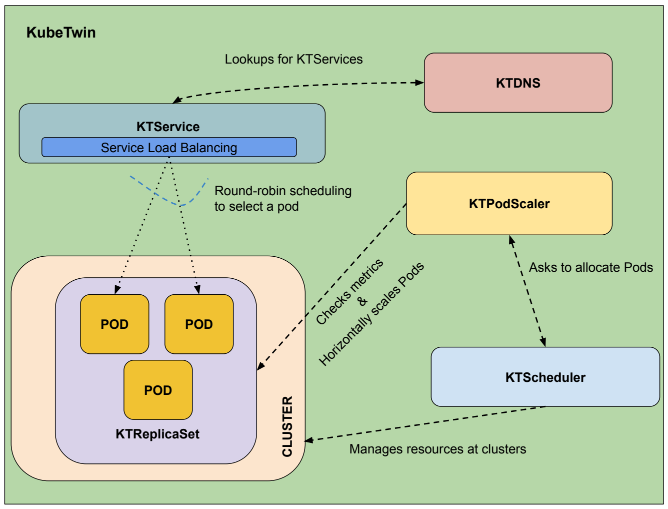

Fig. 2. Interactions between KubeTwin components.

On the other hand, Fig. 2 shows the KubeTwin Domain Name Service (KTDNS), which implements the naming resolution and lookup functionality within KubeTwin. Specifically, KubeTwin registers all KTServices into KTDNS using a label as the unique identifier for a KTService. Once registered, KubeTwin components can query the KTDNS to retrieve information about active KTServices.

Then, the KTReplicaSet visible in Fig. 2 resembles the functionality of the Kubernetes ReplicaSet component. KubeTwin users need to define a KTReplicaSet for each KTService to specify the number of pods associated with the KTService. Precisely, KubeTwin models a KTReplicaSet using two parameters: a selector for identifying the corresponding KTService and the number of associated replicas. At the beginning of the simulation, each KTReplicaSet instantiates the specified number of replicas, then it continuously monitors their status to activate new ones when necessary, e.g., following the crash of a pod.

Regarding the models for the computation elements, we define a container as a single unit of execution. Each container implements a single software component describing the amount of CPU and memory required. Using a recurrent choice in the scientific literature, which also well suits the simulative approach [18], we use a G/G/n/FIFO queuing model to reenact the process of serving requests at the software component level. More specifically, KubeTwin users can configure the level of parallelism, the maximum queue size, and the request service times associated with a specific software component - in the latter case, by defining a random variable from which KubeTwin will sample the service times. KubeTwin provides a wide range of well-known distributions, from exponential to log-normal to Pareto-family ones, but also allows the use of empirical distributions obtained from real-life measurements. This design choice gives KubeTwin users significant freedom to explore model accuracy against complexity and bias against variance tradeoffs, within a conceptual framework that is well-understood and relatively easy to work with.

We also assume that a pod hosts a single container. This design choice simplifies the design of KubeTwin while still allowing us to accurately model the overwhelming majority of Kubernetes applications, which adopt the “one container per pod” policy (usually considered a best practice). However, we intend to support multi-container pods in future versions of KubeTwin to enable the accurate reenactment of the small share of applications that leverage pods with multiple affine containers (typically a main container and one or more “sidecar” containers).

The KubeTwin Scheduler (KTScheduler) plays a relevant role within KubeTwin. It is responsible for managing computing resources according to a set of configurable policies, including the result of automated scaling procedures. Specifically, KTScheduler selects the computing node where to activate a new pod considering the pod’s requirements, the status of available computing resources, and the configured scheduling policy [19]. A three-step procedure regulates the selection of a computing node. Firstly, KTScheduler filters the resources of available clusters to retrieve a list of computing nodes with a residual computing capacity to fit the new pod. Secondly, the KTScheduler assigns a score to the filtered nodes following a configurable scheduling policy. Finally, KTScheduler selects the node with the highest score as the candidate for deploying the pod. To this end, it is worth noting that the KTScheduler executes the filter-and-score procedure each time there is a request for a new pod.

KubeTwin users can tune the KTScheduler behavior by providing their own scoring policy, thus allowing them to experiment with custom resource schedulers for Kubernetes in reproducible environments - a very hot research topic [20]. Interesting yet simple examples of scoring policies could be sorting the filtered nodes according to the highest available residual capacity, thus leading to relatively evenly distributed resource allocations that maximize the responsiveness of autoscaling processes, or the lowest renting prices to minimize the overall service provisioning costs. A different, slightly more complex, scoring policy could be to assign a higher priority to the nodes located in the proximity of users' premises. This could be a reasonable choice for MEC applications with low latency requirements.

Finally, let us specify that while the configuration of these components might be slightly different from the one used in Kubernetes, it is easy to translate KubeTwin-specific configuration into Kubernetes and vice versa. This translation effectively implements a bidirectional link between the digital twin running on the top of KubeTwin and the Kubernetes deployment.

B. Automated Scaling

The components described in the previous Section allow KubeTwin users to simulate a static application deployment. However, the power of Kubernetes also resides in its automated scaling solutions. Among those, the Horizontal Pod Autoscaler (HPA) is a well-adopted solution that Kubernetes can leverage to scale a deployment according to the current workload.

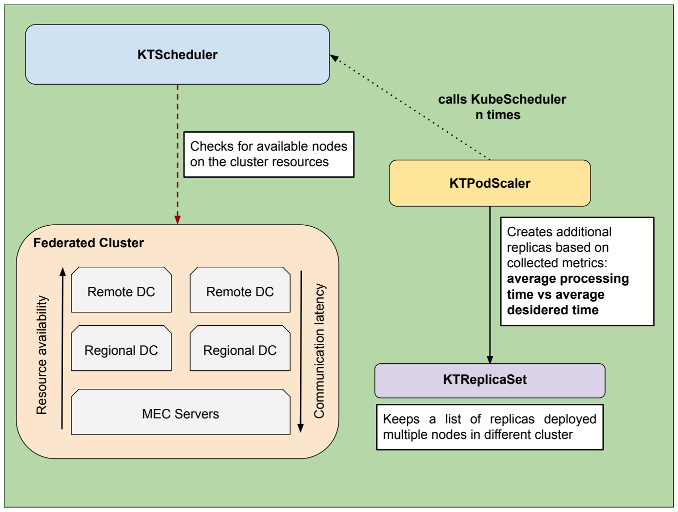
KTScheduler
calls KubeScheduler n times
Checks for available nodes on the cluster resources
KTPodScaler
Creates additional replicas based on collected metrics: average processing time vs average desidered time
Federated Cluster
Resource availability
Remote DC
Remote DC
Regional DC
Regional DC
MEC Servers
Communication latency
KTReplicaSet
Keeps a list of replicas deployed multiple nodes in different cluster

Fig. 3. The main operations performed by KTPodScaler for allocating pods on the available computing nodes in the federated cluster.

To reenact autoscaling behavior, KubeTwin provides the KubeTwin Pod Scaler (KTPodScaler) component, as illustrated in Fig. 3. KTPodScaler is implemented as a periodic control loop to check the current performance of the associated KTReplicaSet, at configurable time intervals. According to the HPA specification [21], KTPodScaler monitors the performance of the application by calculating the average processing time of the associated KTService, considering all instantiated replicas. If the current processing time $T_{\text{proc,current}}$ is higher than the expected value $T_{\text{proc,desired}}$ (i.e., the current number of replicas cannot handle the number of requests) the KTPodScaler will immediately try to activate new ones. To avoid the under/over scaling of application components, KubeTwin users must specify the values for $maxReplica$ and $minReplica$ parameters, which set an upper and lower bound for the pod replicas associated with a KTReplicaSet.

More specifically, KTPodScaler calculates the number of replicas as follows:

$$N_{\text{replicas}} = \left\lceil N_{\text{replicas,current}} \times \frac{T_{\text{proc,current}}}{T_{\text{proc,desired}}} \right\rceil \tag{1}$$

where N_{replicas} should not exceed the specified $maxReplica$ parameter and be lower than the $minReplica$ parameter. To specify the $T_{\text{proc,desired}}$ parameter, KubeTwin users can set a tolerance range to indicate an interval within which the time to process a certain request type is acceptable. For example, KubeTwin users may decide to maintain the same number of replicas if $T_{\text{proc,current}}/T_{\text{proc,desired}} \in [0.9, 1.10]$, i.e., a 10% tolerance range.

Finally, to give a numerical example of Eq. (1), let us consider the case in which there are 20 running replicas for an image processing service component, which is expected to process a request in less than 8ms, while the current average processing metric is 15ms. In this case, the KTPodScaler would have to instantiate new replicas to speed up the processing for the service component. Applying the formula in Eq. (1), we have that the number of replicas should be increased to $N_{\text{replicas}} = \lceil 20 \times (15/8) \rceil = 38$. This behavior adheres with the Kubernetes-specific implementation described in [21].

To decide where to activate the new replicas, the KTPodScaler interacts with KTScheduler, which runs the filter-and-score procedure to select the computing nodes. Let us note that the KTPodScaler works in both ways, i.e., to increase or decrease N_{replicas} according to the current workload to avoid a waste of computing resources.

It is worth noting that a KTPodScaler is KTService-specific. This means that KubeTwin users can choose whether to create a KTPodScaler for each KTReplicaSet of their application. For example, in a complex microservice application, only some software components may be associated with the auto-scaling feature, while others could not.

Finally, let us note that the evaluation of the performance of application autoscaling within KubeTwin represents an important link from the virtual representation of a Kubernetes application to its real-life counterpart. In fact, when KubeTwin identifies well-performing values (or value ranges) for KTPodScalers configuration parameters (i.e., an optimal number of replicas for a given component), it sends that information to the real Kubernetes deployment to be used for priming the autoscaling configuration – allowing it to work in a proactive fashion or even with a predictive process.

C. Communication Model

Following the approach we used in [7], we model the latency values between software components using random variables. This approach is far more accurate than other approaches considering either static communication latency values or Euclidean distances between two distinct locations. Instead, the use of random variables enables the modeling of more realistic communication latency values, which could be sampled from historical data distributions or purposely defined distributions to test, for instance, the effect of increased latency values in communication links and so on. Specifically, these random variables are completely configurable by KubeTwin adopters, including the case of random distributions generated from latency values measured in real distributed computing environments, to allow a realistic reenactment of different use cases. This approach enables the specification of distinct latency models for pods running within the same cluster ("intra-cluster communications") and pods running in separate ones ("inter-cluster communications"). Therefore, it increases the realistic degree of the simulation and lets users specify complex deployment scenarios such as Edge-Cloud deployment, in which some of the software components are distributed at the edge of the network, while others are running in cloud computing facilities.

Delving into implementation details, KubeTwin employs "locations" to identify different communication endpoints, i.e., data centers in distinct geographical locations. The latency between these locations is specified through a matrix notation with elements $RV(i,j)$. Each element $RV(i,j)$ is a random variable modeling the latency between locations i and j. Therefore, KubeTwin calculates the network delay for a given pair of locations i, j by sampling from the corresponding $RV(i,j)$. Finally, let us specify that communication latency between two endpoints can be symmetric $RV(i,j) = RV(j,i)$ or asymmetric $RV(i,j) \neq RV(j,i)$. In the former case, the matrix would be symmetric, while in the latter one, the matrix would be a square matrix.

IV. Methodology Validation

Following MEC architectural principles defined in [14], Kubernetes could be a good candidate to host MEC applications. In detail, a Kubernetes cluster could serve as the means to offer the Virtualization Infrastructure capabilities inside a MEC Host, thus allowing the deployment of MEC applications as Kubernetes workloads. To accurately reenact these Kubernetes workloads using digital twin methodologies, there is the need to compute an accurate model that describes the different parts of a Kubernetes environment, such as the statistical distribution of the microservice processing times, the communication latency values between different locations, and the distributions of service requests.

In a previous article [12], we demonstrated how the response time of microservices could be modeled with a simulation-based inference procedure. In detail, we took a candidate application composed of two microservices and collected the metrics of their response time with increasing incoming request rates and with different numbers of microservice replicas. With this data, we computed the statistical distribution of the microservices as a Gaussian Mixture Model (GMM) using a Quantum-inspired Particle Swarm Optimization (QPSO) algorithm. Finally, we plugged these models into the simulator to reenact particular system working conditions and compared them with the results obtained from the simulated system.

In [12], we proved that the presented methodology could reenact the behavior of a container-based application with a reasonable degree of accuracy, thus making it suitable for defining digital twins of Kubernetes environments. Therefore, in this manuscript, we instead focus on presenting the capabilities that a framework such as KubeTwin can provide to its adopters. Specifically, we will extend the types of scenarios considered by introducing descriptions and models for intra and inter-cluster communication delays. Furthermore, building from the microservice description of [12], we will show how the developed simulator could be employed for running what-if scenario analysis.

V. Use Case

As a representative use case to showcase the potential of KubeTwin, we present a container-based image recognition application for which we model its Kubernetes digital twin. An image recognition application is a well-suited service for MEC applications. In fact, such a service could be a building block for a variety of virtual and augmented reality applications, which rely on image recognition services for different tasks [13]. Therefore, we believe the image recognition application represents an interesting case study of a machine learning algorithm running as a MEC service that interested users can interact with using their User Equipment.

The effective implementation of such next-generation applications calls for strict latency and computing requirements necessary to provide a prompt response to the users requesting the service. We assume that MEC plays a significant role in enabling the provisioning of such applications [22]. Computation offloading at the edge provides enhanced QoE to the users by reducing the communication and processing delay, e.g., the processing is offloaded to dedicated MEC servers located near the access part of the network.

In this Section, we describe the use-case application that we implemented as a container-based Kubernetes application. In addition, we discuss how to create a digital twin model for the application that can be used for evaluation in KubeTwin. Finally, we provide an accurate description of a Compute Continuum scenario with MEC and cloud resources where this and similar applications can be deployed.

A. Description

The use-case application implements an image-recognition algorithm as a two microservices service chain. The first microservice (MS1) resizes the images contained within the user requests, while the second microservice (MS2) implements the object recognition algorithm and relays the information back to MS1. To enable network communications, both microservices implement a RESTful API built upon the Flask Python framework.

To deploy the application on Kubernetes, we define a container image for MS1 and MS2 and the relative deployment and NodePort configuration files. Then, for setting up a Kubernetes cluster testbed, we leverage three Virtual Machines (VM) with four vCPU cores and 8GB of RAM each, running a standard installation of Kubernetes with Calico as Container Network Infrastructure (CNI). We deploy the two microservices as two different containers running in separate pods. Furthermore, we include a ReplicaSet configuration to manage the number of replicas associated with each microservice. Finally, we leverage the described Kubernetes deployment to compute the statistical distribution of the microservices processing time using the methodology described in Section IV.

B. Inter-cluster Communication Model

To model the inter-cluster communication latency for the use case that considers both edge and cloud computing resources, we assumed the availability of computing facilities in different locations. We classified those locations according to the relative distance from the end user, following an approach similar to what the Linux Foundation Edge (LFE) initiative proposes [23]:

- *Local DC*: computing facilities located at the customer premises, within an average latency range $d < 10$ ms, corresponding to LFE “Service Provider Edge” or “Access Edge”;
- *Tier 1 Regional DC*: computing facilities located close to the customer premises, within an average latency range $10 \leq d < 20$ ms, corresponding to LFE “Tier 1 Regional Edge”;
- *Tier 2 Regional DC*: computing facilities located close to the customer premises but not as close as Tier 1 Regional DC, within an average latency range $20 \leq d < 40$ ms, corresponding to LFE “Tier 2 Regional Edge”;
- *Remote DC*: computing facilities located in a remote cloud, within an average latency value $d \geq 100$ ms, corresponding to LFE “Centralized DC”.

The latency among these computing facilities was modeled with random values generated from latency measurements performed over TCP connections between multiple Amazon Web Services (AWS) data centers and available on the CloudPing website.[2] The CloudPing measurements are averaged over a one-year period and provide mean values as well as several percentile values. We limited our choice to a subset of the AWS data center locations available at CloudPing, based on reasonable assumptions made on the measured latency value ranges. In particular, we chose:

- as Local DC the *eu-south-1* data-center, located in Milan, Italy;
- as Tier 1 Regional DCs the *eu-central-1* data-center located in Frankfurt, Germany and the *eu-west-3* data-center located in Paris, France;
- as Tier 2 Regional DCs the *eu-west-2* data-center, located in London, UK, and the *eu-north-1* data-center, located in Stockholm, Sweden;
- as Remote DCs the *ca-central-1* data-center, located in Canada, and the *us-east-1* data-center, located in Virginia, US.

To randomly generate the latency values for each pair of computing locations in the KubeTwin simulations, we adopted a Gaussian distribution with mean μ and variance σ^2, truncated to the interval $[d_{\min}, d_{\max}]$. However, since the CloudPing website does not provide the complete statistics of the measured latency values, we had to compute the standard deviation from the average and percentile values available. To do so, we recall that for a Gaussian distribution with mean μ and variance σ^2 the p-th percentile value x_p is such that:

$$\text{Prob}\{x \leq x_p\} = F_N(x_p) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{x_p - \mu}{\sqrt{2}\,\sigma}\right)\right) = p \tag{2}$$

where F_N is the Gaussian cumulative distribution function and $\text{erf}(x)$ is the error function. Solving (2) for σ we obtain the standard deviation from a given percentile value:

$$\sigma = \frac{x_p - \mu}{\sqrt{2}\,\text{erf}^{-1}(2p - 1)} \tag{3}$$

Starting from the mean values and the 99-th percentiles available on the CloudPing website, we calculated μ and σ values for each inter-cluster pair as reported in Table I, which we also report as a reference for readers and other researchers interested in modeling similar scenarios.

C. Intra-cluster Communication Model

Before delving into the analysis of intra-cluster pod communications, let us clarify the network setup of containers in Docker. Docker and Kubernetes are completely interoperable, anyone can use Docker without Kubernetes and vice versa, however, they work well together in several scenarios including Industry 4.0 and IoT applications. For this specific communication model, we have taken into account two different network setups in Docker: i) *bridge* that is the default network in Docker where each container has its own network namespace; ii) *host* that does not create a separated network stack for containers, instead, each container shares the same network stack with the host. Since a pod consists of one or more containers that are located on the same host, they can share the network stack or not. See Fig. 4 for a visual representation.

We measured the latency over the network between two containers in both network modes (i.e., bridge and host). In particular, to measure latency over the network, we have used

[2]https://www.cloudping.co/

TABLE I
INTER-CLUSTER GAUSSIAN CONFIGURATION PARAMETERS (VALUES EXPRESSED IN MS)

	eu-central-1	eu-west-3	eu-west-2	eu-north-1	ca-central-1	us-east-1
eu-south-1	$\mu = 12.60$ $\sigma = 5.82$	$\mu = 19.80$ $\sigma = 1.95$	$\mu = 26.58$ $\sigma = 2.05$	$\mu = 32.28$ $\sigma = 6.43$	$\mu = 104.55$ $\sigma = 6.73$	$\mu = 100.13$ $\sigma = 1.89$
eu-central-1		$\mu = 10.53$ $\sigma = 2.13$	$\mu = 16.34$ $\sigma = 1.82$	$\mu = 23.16$ $\sigma = 3.18$	$\mu = 94.61$ $\sigma = 6.46$	$\mu = 91.08$ $\sigma = 2.05$
eu-west-3			$\mu = 10.27$ $\sigma = 1.93$	$\mu = 30.58$ $\sigma = 3.92$	$\mu = 87.64$ $\sigma = 6.19$	$\mu = 83.41$ $\sigma = 1.75$
eu-west-2				$\mu = 32.81$ $\sigma = 2.19$	$\mu = 80.73$ $\sigma = 6.13$	$\mu = 78.12$ $\sigma = 6.40$
eu-north-1					$\mu = 110.59$ $\sigma = 7.08$	$\mu = 108.56$ $\sigma = 2.69$
ca-central-1						$\mu = 17.66$ $\sigma = 6.55$

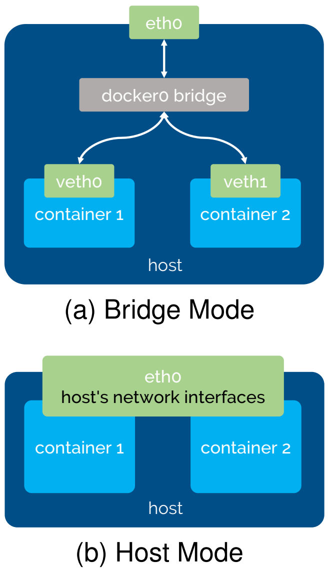

Fig. 4. The difference in architecture between bridge networking mode and host networking mode in Docker

TABLE II
LATENCY COMPARISON BETWEEN THE DOCKER BRIDGE AND HOST MODES

	Bridge		Host	
Payload (bytes)	median (µs)	std (µs)	median (µs)	std (µs)
1	47.50	6.66	43.37	5.11
2	47.50	6.62	43.38	4.17
4	47.51	6.33	43.38	4.66
8	47.52	5.80	43.38	4.62
16	47.52	6.44	43.38	5.11
32	47.52	5.86	43.38	4.04
64	47.53	5.92	43.39	4.39
128	47.62	5.59	43.51	4.30
256	47.64	5.40	43.53	5.07
512	47.68	4.97	43.56	4.06
1024	47.97	4.92	43.88	4.55
2048	52.88	5.45	44.00	4.70
4096	55.70	5.81	44.29	4.80
8192	56.42	5.96	44.94	4.88
16384	57.56	6.28	46.06	4.48
32768	59.75	6.86	48.33	5.32

the software *sfnt-pingpong*[3] that measures ping-pong latency over a range of message sizes by using standard network protocols including TCP and UDP. One container operates as the client and the other as the server. The output identifies the median (microseconds) RTT/2 latency for increasing TCP packet sizes, including the standard deviation for these results. Results are reported in Table II for both bridge and host modes.

Considering the above measurements and that the Kubernetes network layer usually operates in bridge mode, in this work we decide to model the intra-cluster communication latency considering a multimodal packet distribution with a payload size of 32, 128, and 1024 bytes, also considering a similar characterization in [24]. Therefore, we use the corresponding mean and standard deviation values illustrated in Table II to define three random variables with a Gaussian distribution. Finally, we construct a Gaussian mixture random variable for intra-cluster communication as the summation of the three Gaussian random variables as follows:

$$\Sigma_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) \tag{4}$$

where each random variable $\mathcal{N}(x|\mu_k, \Sigma_k)$ has the same weight $\pi_k \approx 0.33$, and $\Sigma_{k=1}^{K} \pi_k = 1$. We use this mixture as a sufficiently flexible initial approximation for modeling intra-cluster latency, while a more accurate fit of the weights π_k is a topic of separate research. Different configurations of parameters π_k, μ_k, Σ_k are to be determined and selected depending on the specific scenario requirements.

VI. EVALUATION

To evaluate KubeTwin, we devised a set of experiments on the image recognition application described in SectionV-A. The main objective of these experiments is to validate the KubeTwin simulation framework as a valuable solution to experiment with different Kubernetes deployments. To this end, we first present a performance comparison between several simulations using different deployment configurations to find a suitable deployment for the use-case case described above. Then, we show a second experiment in which we validate horizontal scaling reenactment within the KubeTwin framework.

[3]https://github.com/Xilinx-CNS/onload

A. Deployment Scenario

We envision that the image-recognition service will be available to users located in the smart city of Milan, in which a MEC DC is available to its citizen. Fig. 5 illustrates the MEC deployment in a federated cluster scenario which we considered for the experimental evaluation. Specifically, we have a small-size MEC DC in Milan, which we model as a single Local DC in the proximity of the user premises, that provides computing capabilities at a reduced communication latency, two Tier 1 Regional DCs, and two Tier 2 Regional DCs. We also model Cloud clusters that can be beneficial to scale the application performance when needed, e.g., when the resources at lower tiers are saturated, by providing enhanced and on-demand computing capabilities at the price of higher latency. We identify these resources as the data centers flagged in the Remote DC tier. The assumption that users are located in the proximity of the Local DC is a typical example of a small edge data center located at the smart city borders. There, users play with some “augmented reality application”, which continuously calls the image recognition service to identify objects within the camera frames collected using the application. Communication latency between the different locations is modeled according to the models presented in Sections V-B and V-C.

With regard to the computing capabilities at the different tiers, we model the availability of 25 nodes in the MEC DC, 100 nodes in Tier 1, 150 nodes in Tier 2, and 200 nodes in the Remote DC tier. For the following experiments, we assume that the computing nodes at Local DC, Tier 1, and Tier 2 have the same amount of CPU and memory resources, which correspond to a constant value of 100 for both CPU and memory. Such value refers to a standard amount of resources required by a machine with medium computational power (i.e., an i7 core). Instead, we assume that computing nodes in the Remote DC are twice as powerful, i.e., they have a value of 200 for both CPU and memory. This metric allows us to have a good granularity in modeling the computational capacity gap between the different layers of the cluster and to evaluate the behavior of Kubernetes applications in a Compute Continuum scenario, in which the availability of more powerful computing nodes is typically offset by increased communication latency.

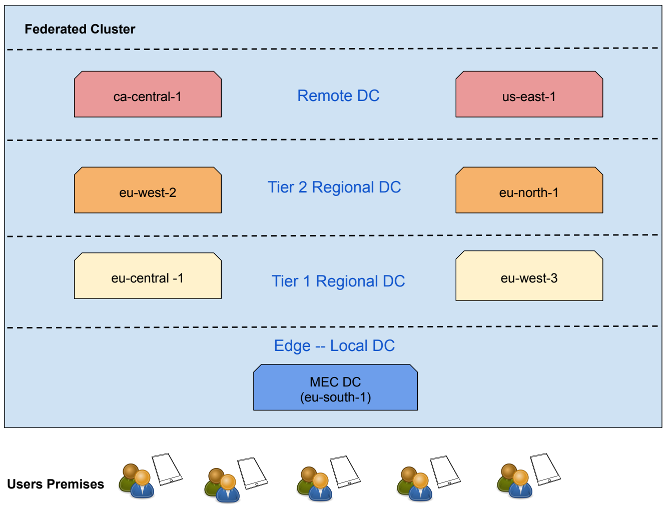

Fig. 5. The federated cluster scenario described for the experiments. Users are located in the proximity of the MEC DC, thus can benefit from a reduced communication latency. Other tiers provide computing nodes to distribute the application load.

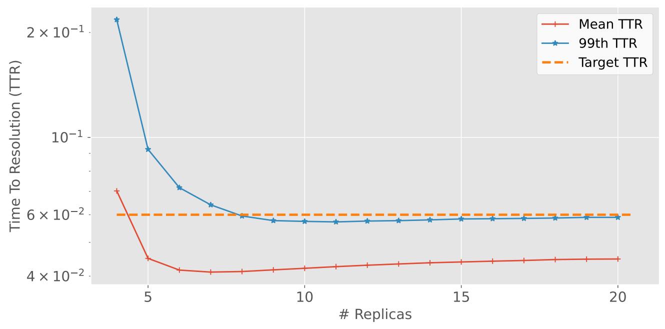

Fig. 6. Mean and 99th percentile TTR at different numbers of replicas

Finally, at the scheduling level, we assign higher scheduling priorities (node affinity) to the computing nodes residing at the lower tiers. Specifically, nodes at the Local DC have the maximum scheduling priority, then each tier at a higher level presents a decreasing scheduling priority. This choice defines a MEC-first filter-and-score procedure that will make the KTScheduler select computing nodes close to the user premises first. Let us note that this scheduling priority is different from the default one, which would try to equally distribute the replicas among all available nodes in the cluster, regardless of the performance in terms of latency.

B. Identifying Best Deployment Configuration

For the first validation, we show how KubeTwin can be used to identify a suitable configuration for the scenario of interest. Specifically, with this analysis, we need to identify the number of replicas associated with each microservice. This procedure can be used to find a suitable deployment configuration that could satisfy the requirements of a relatively static scenario, which is characterized by a constant workload of user requests. Furthermore, we want to demonstrate the effectiveness of KubeTwin as a digital twin, whose purpose is to help service providers to identify a suitable configuration for their real-world system.

For the following experiments, we model the user activity as an aggregated flow of requests with a constant intensity. More specifically, we define the interarrival times by sampling from a random variable with exponential distribution to simulate a workload of 100 requests per second (rps). Moreover, we set the request generator to send a total of 10,000 requests, considering 10 seconds for the simulation warmup and 10 seconds for the simulation cooldown. Each microservice has a dedicated KTReplicaSet, which maintains a stable set of replica pods running at any given time.

Our objective is to evaluate different deployment configurations by analyzing the *Time To Resolution (TTR)*, i.e., the time elapsed between the receipt of a request message and the emission of a corresponding response message for the

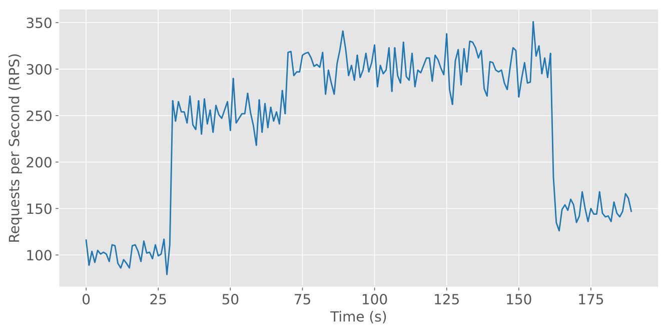

Fig. 7. The distribution of the requests during simulation time.

application workflow during the simulation time. To this end, we ran multiple simulations by setting different numbers of replicas (from 1 to 20) and collecting the TTR of each request to calculate the mean and the 99th percentile TTR. For the sake of clarity, these values (i.e., mean and 99th percentile TTR) include the time spent by requests while traveling to the simulated locations (i.e., the time spent transmitting the request between users and data centers). With this experiment, we could visually identify those deployments capable of fulfilling a given Service Level Indicator (SLI). In this case, we set a target 99-th percentile TTR of 60 ms and we look for a deployment configuration capable to satisfy the 60ms SLI.

The results of this validation are visible in Fig. 6, which shows the TTR and 99th percentile TTR while varying the number of replicas on the x-axis. Let us note that we excluded from Fig. 6 the results for the configurations with less than four replicas, which were out of scale. Specifically, looking at Fig. 6, it is easy to note that configurations with up to 8 replicas struggle to meet the application requirements (60 ms 99th percentile TTR) even if the mean TTR is below the threshold, since the 99th percentile TTR of the requests is still above the given target. On the contrary, the configurations with 9 and more replicas show a decreasing trend in both the mean and the 99th TTR. Furthermore, Fig. 6 also shows that over scaling the number of replicas does not improve the performance in terms of mean and 99th percentile TTR. Therefore, we can visually determine that a deployment configuration with 9 replicas for each microservice (MS1 and MS2) can fulfill the stringent 60 ms SLI. Additional replicas could be instantiated to tackle temporary fluctuations in the application workload. For this reason, we select a deployment configuration with 10 replicas for both MS1 and MS2.

As another interesting result, let us note that considering the small number of pods to allocate, the edge computing resources at the Local DC are sufficient to deploy up to 20 replicas for each microservice. Therefore, KubeTwin can effectively reenact also custom Kubernetes scheduling policies, such as the specified one that tries to instantiate all pods in the MEC DC first. This would allow service providers to estimate the deployment costs for multiple configurations, e.g., shifting from the different tiers of the Compute Continuum depending on the prices of the computing resources but also on the specific requirements of these applications.

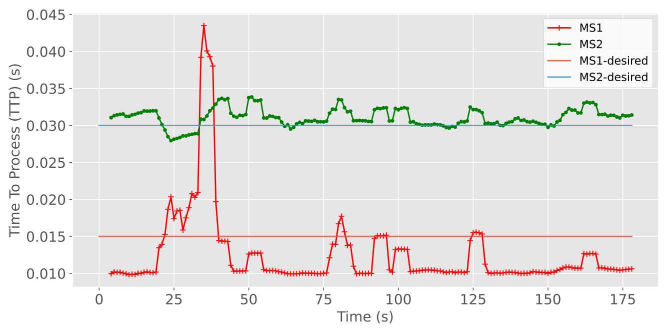

Fig. 8. Average Time-to-Process (TTP) requests per micro-service during the simulation time compared to the desired TTP.

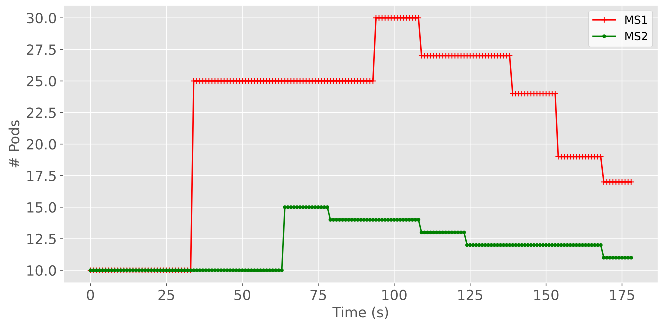

Fig. 9. Distribution of the number of pods for each microservice during the simulation time.

C. Validating KTPodAutoscaler

In this second validation, we would like to verify if KubeTwin can support horizontal pod autoscaling. Starting from the previous experiment, we define a more workload-aggressive scenario. More specifically, for this experiment, we characterize the aggregated flow of user requests as the summation of several flows of constant interarrival times to stress the performance of the current deployment. This experiment should validate if KubeTwin can increase the number of replicas to meet the increased workload. It is worth noting that this kind of experiment, which stresses the application by applying an increasing workload, can find many employments. For example, a service provider might need to test its Kubernetes deployment to configure the lower and upper bound for the Kubernetes HPA.

For this scenario, we reenact the processing of 130,000 requests. Specifically, we kept a baseline workload of 100,000 requests at 100 rps, which was then increased after 30 seconds since the beginning of the simulation by an additional workload of 20,000 requests at 150 rps. Finally, we simulated an additional 10,000 requests at 50 rps starting 70 seconds after the beginning of the simulation. For the sake of clarity, we illustrate the workload distribution in Fig. 7, which shows a spike of requests starting from the 25th second of the simulation and ending around the 160th second of the simulation, when the workload of 20,000 requests at 150 rps terminates. This workload pattern defines an unexpected spike in the number of requests that would undermine the performance of the current deployment configuration.

To reenact autoscaling features, we specify in the KubeTwin configuration a KTPodScaler component for each microservice that would periodically check the desired performance scale of our targets. In addition, for each KTPodScaler we define the parameters for $minReplicas = 10$ (the deployment configuration selected at the previous step) and $maxReplica = 50$, and a period of 15 seconds (the default interval according to Kubernetes documentation). From the most basic perspective, the KTPodScaler analyzes the ratio between the desired metric value and the current metric value (see Eq. 1), skipping any scaling action if the ratio is within the $[0.90, 1.10]$ interval.

We report the results of the validations in Fig. 8 and Fig. 9, which show respectively the *average* Time To Process (TTP) along with the *desired* TTP and the number of instantiated pods measured during the simulation time. Specifically, the average TTP indicates the average time for processing a request, calculated as the difference between the finished processing time and the time at which the request arrived at the container, whereas the desired TTP for each microservice is defined as its expected processing time (i.e., when the system is not overloaded) plus a 20% tolerance, which is the maximum value that we are willing to accept.

Figure 8 shows that the most affected microservice is MS1, which is the entry point for the application workflow. Specifically, we can see a notable increase in the TTP, which is mainly due to a non-optimal number of instantiated replicas and a very strict desired metric – illustrated with straight orange (MS1) and blue (MS2) lines in the figure. On the other hand, MS2 is less affected by the increased workload, as it starts processing each request only after MS1 finishes the initial processing. Therefore, the TTP trend for MS2 is almost linear with some increase during the simulation time.

To improve the performance of the current configuration, the KTPodScalers intervene to reduce the TTP for MS2 and MS1 by allocating more replicas, as visible in Fig. 9. Specifically, this happens around the 30th second of the simulation, i.e., in correspondence with the increased workload, where the TTP starts to increase for all microservices. This behavior happens because the current deployment (10 pods for MS1 and 10 pods for MS2) cannot process the increased amount of incoming requests, i.e., these requests get queued. However, as soon as the KTPodScalers intervene to activate new replicas, the TTP for the illustrated microservices starts to decrease, stabilizing for the rest of the simulation. The activity of the KTPodScalers is also visible in Fig. 9, which shows that the number of associated pods changes to cope with the increased request workload and to respond to performance degradation. Let us note that the number of instantiated pods for MS1 goes up to 30, and the one for MS2 goes up to 15 to address the notable increase in the TTP, while it gradually decreases when the workload goes back to 150 rps as expected. Additionally, let us specify that while the average TTP illustrated in Fig. 8 is the result of all requests processed during simulation time, the KTPodScalers take the scaling decisions by considering the metric calculated over 15 second intervals, as described in Section III.

Finally, let us also analyze the distribution of the instantiated replicas for the most intensive deployment, retrieved from the seconds 105th to 120th of the simulation time. During this time window, the KTPodScaler instantiated 30 pods for MS1 and 14 pods for MS2 among the Local DC and the Regional DC at Tier 1. Specifically, this distribution is the result of the scheduling policy that tries to exploit computing resources at the lowest tiers first. It is worth noting that this scheduling policy would define deployment more performant in terms of TTR because of the reduced communication latency given by the proximity of these computing resources to end-users.

D. Discussion

We believe that the illustrated results demonstrate the soundness of the KubeTwin framework as a digital twin for Kubernetes deployments. Let us specify that, even if it is still in the development stage, KubeTwin can provide helpful guidance to service providers and Kubernetes adopters who want to experiment with different deployment configurations. Furthermore, we would like to point out the realistic evaluation of KubeTwin, which instead of adopting an analytical model for simulating the processing of requests, reenacts the execution of every user request, starting from its generation to their arrival at the user premises.

Therefore, we believe that KubeTwin can be highly beneficial in tuning the desired behavior of Kubernetes applications. In fact, digital twin methodologies enable running what-if scenario analysis and then evaluating different deployment opportunities depending on available pricing alternatives, resource availability, scheduling policies, and the current demand of users. Let us also note that this exploration process could be automatized to evaluate several configurations in a shorter time, thus allowing the digital twin to predict the proper configurations in case of environmental changes (e.g., increased communication latency and increased user demand). When such changes are detected, KubeTwin can then elaborate a new configuration and then instantiate it into the Kubernetes application, thus realizing the bidirectional link between the virtual and the physical elements of the DT.

As a final consideration, let us discuss the time required for running an analysis with KubeTwin, which mainly depends on the characteristics of the simulation scenario and the number of requests to process. We ran the above experiments on a 2019 Dell XPS with a 6-cores Intel Core i7 2.60GHz CPU, 32 GiB RAM, and a 64-bit version of Linux Manjaro (5.15.114-2) configured with a Ruby MRI interpreter (version 3.0.2). The execution of the first experiment took about 2.72 seconds to simulate the processing of 10,000 requests. Instead, the second experiment took roughly 8.85 seconds to simulate 120,000 requests. Considering that KubeTwin is still not optimized for performance-wise usage, these results highlight the most compelling advantage of running what-if scenario analysis with KubeTwin, i.e., by facilitating accurate simulations, KubeTwin drastically reduces the computing time required for testing, compared to conducting these tests on physical Kubernetes deployments.

VII. Related Work

Recently, there has been growing attention from the research community for the adoption of Kubernetes as a network and service management solution. On the one hand, some works investigate its adoption as an orchestration solution for the management and orchestration of services running in edge and cloud environments. On the other hand, researchers have started to look into Kubernetes as the orchestration platform for managing container-based Virtualized Network Functions (VNFs) in 5G and beyond network scenarios, complementing established solutions based on Open Source MANO.

Among the related efforts, the work in [25] studies deep learning models to horizontally and vertically scale VNFs in multi-domain networks. Specifically, the authors investigate centralized and distributed learning approaches and verify their effectiveness using a network operator dataset for training and validation. Then, the trained model is plugged into an AI-driven Kubernetes orchestration prototype.

In [26], the authors propose a proof-of-concept MEC compliant implementation using Kubernetes and HELM. Chaudhry et al. present a Kubernetes-based approach that leverages serverless computing to integrate MEC and NFV at the system level and deploy VNFs on-demand in [27]. The authors in [28] present the design of an SFC controller for Kubernetes that optimizes the placement of service chains in Fog environments. Scazzariello et al. introduce an ETSI NFV compliant architecture called Megalos that leverages Docker and Kubernetes for the realization of VNFs and their orchestration of nodes in [29].

While there is a compelling interest in the adoption of Kubernetes, there are very few works that tried to formalize the design of Kubernetes digital twins. Among them, Ghirardini et al. describe an attempt for developing a Kubernetes simulator that could help service developers to understand self-configuring systems in [30]. The proposed solution leverages Eclipse's Palladio-Simulator[4] to reenact Kubernetes's functions and behavior. Another effort is the one in [31], which presents a systematic approach for modeling the performance of microservices in cloud native service chains.

Recently, researchers have started organizing the first events specifically dedicated to digital twins in networking. Among these, we report the 2022 and the 2023 editions of the International Workshop on Technologies for Network Twins (TNT), co-located with IEEE/IFIP Network Operations and Management Symposium (NOMS), a flagship IEEE conference and a reference event for the Network and Service Management research community [32]. The workshop included keynotes from world-leading experts, panels dedicated to the definition of the digital twin concept in network and service management area of applications, and many interesting papers focusing on a wide range of applications such as data center health optimization [33], traffic reduction [34], chaos engineering [35], and what-if analysis in BGP optimization [36]. Alongside this kind of event, a strong interest in adopting Network digital twins is emerging. In [37], the authors include Machine Learning-based digital twin in a Service Function Chain Platform definition to obtain training samples more conveniently and affordably than collecting them from a real system. Thanks to this approach, they gain much better performance in terms of throughput and latency in their Virtual Network Functions optimization goal.

[4]https://www.palladio-simulator.com/home/

More in general, digital twin approaches find their application also to related fields, such as smart manufacturing and healthcare. Groshev et al. propose the digital twin as a Service (DTaaS) concept and present the case study of an Edge Robotics digital twin system in [38]. More specifically, the authors analyze the capabilities of the digital twin system to provide potential savings by considering different computational offloading models and the impact of different wireless channels (5G, 4G, and WiFi) on the synchronization between the digital twin and the physical device. In [39], the authors present a digital twin framework for health and well-being in smart city environments. The proposed framework aims to collect data from heterogeneous health devices to improve the quality of smart healthcare services in smart cities. Another effort is given in [40], where the authors discuss the implementation of a building digital twin by leveraging the data collected from IoT sensor networks. The idea is to create a virtual replica of a building facade, which can provide useful findings in determining the arrangement of sensors. In [41] authors present a digital twin Healthcare (DTH) in order to solve the problem of real-time supervision and accuracy of crisis warning for the elderly in healthcare services thanks to technologies like big data, cloud computing, and Internet-of-Things.

In [42], authors introduced a digital twin architecture reference model for cloud-based Cyber-Physical Systems (CPS) called C2PS, which enables the definition of a digital twin by specifying the analytical properties of a CPS. The authors highlight how a digital twin can analyze the current status of the real system to suggest actions that can improve the performance of the real system. As a showcase for their reference model, the authors present an interesting vehicular driving assistance application. Finally, in [43] the authors propose a digital twin for "smart manufacturing" scenarios including cloud, fog, and edge computing domains. The functions of the digital twin are distributed in the different computing domains, taking into account different requirements such as communication latency and computing resource requirements. Furthermore, the digital twin maintains virtual models of the physical appliances evolving based on their real-time state. These virtual models are then used to carry on simulations to adjust and control the behavior of the physical resources.

KubeTwin embraces this final approach, by allowing its users to run what-if scenario analyses on the virtual element of the digital twin to identify optimized configuration deployments. Differently, from the previous work in which we focused on the creation of an accurate statistical model to approximate the processing times of container-based applications [12], this manuscript presents a comprehensive overview of the framework and shows the potentials of the tool by evaluating an image recognition application on a complex computing scenario.

VIII. Conclusion

The implementation of digital twins is a compelling and recent avenue for researchers working in the network and service management domain. In fact, the ever-increasing complexity of the management of applications and networks calls for alternative solutions such as digital twins to identify proper deployment configurations. Even if management and orchestration solutions such as Kubernetes emerged, there is still the need, especially for large-scale scenarios, for evaluating deployment configurations upfront.

To contribute to this research avenue, this paper presents KubeTwin, a comprehensive simulation framework that we developed as a guideline for service providers that need to deploy their applications using Kubernetes-like orchestrator tools. KubeTwin proposes to these service providers a simulation platform that they can use to validate different configuration deployments without sustaining the deployment costs.

We evaluated KubeTwin to demonstrate its effectiveness in creating digital twins of Kubernetes-based software deployment by defining a distributed MEC application. The results confirm the validity of the KubeTwin approach, can reenact the execution of a complex system both at the application and the orchestration level.

The soundness of the results we have achieved so far opens interesting future research directions. First, we are planning to investigate solutions that leverage KubeTwin to identify optimized scheduling policies. To this end, we intend to explore both population-based metaheuristics, such as advanced variants of Genetic Algorithms and Particle Swarm Optimization designed for expensive optimization problems, and reinforcement learning-based solutions. In addition, we are planning to leverage KubeTwin to investigate the application of chaos engineering methodology to the virtual representation of Kubernetes applications – a highly innovative approach that presents compelling opportunities in terms of lower costs and barriers to entry.

Acknowledgments

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRPP) of Next Generation EU (NGEU), partnership on “Telecommunications of the Future” (PE00000001 - program “RESTART”), and by the Spoke 1 “FutureHPC & BigData” of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 - NGEU.

References

[1] A. Rasheed, O. San, and T. Kvamsdal, “Digital twin: Values, challenges and enablers from a modeling perspective,” *IEEE Access*, vol. 8, pp. 21 980–22 012, 2020.

[2] I. Errandonea, S. Beltrán, and S. Arrizabalaga, “Digital twin for maintenance: A literature review,” *Computers in Industry*, vol. 123, p. 103316, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361520305509

[3] P. Bellavista, C. Giannelli, M. Mamei, M. Mendula, and M. Picone, “Application-driven network-aware digital twin management in industrial edge environments,” *IEEE Transactions on Industrial Informatics*, vol. 17, no. 11, pp. 7791–7801, 2021.

[4] L. Zhao, G. Han, Z. Li, and L. Shu, “Intelligent digital twin-based software-defined vehicular networks,” *IEEE Network*, vol. 34, no. 5, pp. 178–184, 2020.

[5] M. Balogh, A. Földvári, and P. Varga, “Digital twins in industry 5.0: Challenges in modeling and communication,” in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–6.

[6] Y. Wu, K. Zhang, and Y. Zhang, “Digital twin networks: A survey,” *IEEE Internet of Things Journal*, vol. 8, no. 18, pp. 13 789–13 804, 2021.

[7] W. Cerroni, L. Foschini, G. Y. Grabarnik, F. Poltronieri, L. Shwartz, C. Stefanelli, and M. Tortonesi, “BDMaaS+: Business-driven and Simulation-based Optimization of IT Services in the Hybrid Cloud,” *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 322–337, 2022.

[8] F. Poltronieri, M. Tortonesi, and C. Stefanelli, “Chaostwin: A chaos engineering and digital twin approach for the design of resilient it services,” in *2021 17th International Conference on Network and Service Management (CNSM)*, 2021, pp. 234–238.

[9] W. Wang, L. Tang, C. Wang, and Q. Chen, “Real-time analysis of multiple root causes for anomalies assisted by digital twin in nfv environment,” *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 905–921, 2022.

[10] M. Zhu, R. Kang, F. He, and E. Oki, “Implementation of backup resource management controller for reliable function allocation in kubernetes,” in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, 2021, pp. 360–362.

[11] M. Fogli, T. Kudla, B. Musters, G. Pingen, C. Van den Broek, H. Bastiaansen, N. Suri, and S. Webb, “Performance evaluation of kubernetes distributions (k8s, k3s, kubeedge) in an adaptive and federated cloud infrastructure for disadvantaged tactical networks,” in *2021 International Conference on Military Communication and Information Systems (ICMCIS)*, 2021, pp. 1–7.

[12] D. Borsatti, W. Cerroni, L. Foschini, G. Y. Grabarnik, F. Poltronieri, D. Scotece, L. Shwartz, C. Stefanelli, M. Tortonesi, and M. Zaccarini, “Modeling Digital Twins of Kubernetes-Based Applications,” *Accepted at 28th IEEE Symposium on Computers and Communications (ISCC)*, 2023.

[13] Multi-access edge computing (MEC); use cases and requirements v2.2.1. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/MEC/001_099/002/02.02.01_60/gs_MEC002v020201p.pdf

[14] Multi-access edge computing (MEC); framework and reference architecture. [Online]. Available: "https://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/02.02.01_60/gs_mec003v020201p.pdf"

[15] M. Fogli, C. Giannelli, F. Poltronieri, C. Stefanelli, and M. Tortonesi, “Chaos engineering for resilience assessment of digital twins,” *IEEE Transactions on Industrial Informatics*, pp. 1–9, 2023.

[16] S. Tuli, G. Casale, and N. R. Jennings, “Dragon: Decentralized fault tolerance in edge federations,” *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, pp. 276–291, 2023.

[17] L. Manca, D. Borsatti, F. Poltronieri, M. Zaccarini, D. Scotece, G. Davoli, L. Foschini, G. Y. Grabarnik, L. Shwartz, C. Stefanelli, M. Tortonesi, and W. Cerroni, “Characterization of microservice response time in kubernetes: A mixture density network approach,” in *2023 19th International Conference on Network and Service Management (CNSM)*, 2023, pp. 1–9.

[18] E. Jafarnejad Ghomi, A. M. Rahmani, and N. N. Qader, “Applying queue theory for modeling of cloud computing: A systematic review,” *Concurrency and Computation: Practice and Experience*, vol. 31, no. 17, p. e5186, 2019, e5186 CPE-18-0152.R1. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5186

[19] “Kubernetes: Scheduling framework,” https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/, [Online; retrieved on January 11, 2022].

[20] C. Carrión, “Kubernetes scheduling: Taxonomy, ongoing issues and challenges,” *ACM Comput. Surv.*, may 2022. [Online]. Available: https://doi.org/10.1145/3539606

[21] “Kubernetes: Horizontal pod autoscaling,” https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, [Online; retrieved on January 11, 2022].

[22] Q.-V. Pham, F. Fang, V. N. Ha, M. J. Piran, M. Le, L. B. Le, W.-J. Hwang, and Z. Ding, “A survey of multi-access edge computing in 5g and beyond: Fundamentals, technology integration, and state-of-the-art,” *IEEE Access*, vol. 8, pp. 116 974–117 017, 2020.

[23] “Sharpening the edge: Overview of the LF edge taxonomy and framework,” White paper, Linux Foundation Edge, https://www.lfedge.org/resources/publications/, [Online; retrieved on January 26, 2022].

[24] R. Sinha, C. Papadopoulos, and J. Heidemann, “Internet packet size distributions: Some observations,” USC/Information Sciences Institute, Tech. Rep. ISI-TR-2007-643, May 2007, orignally released October

2005 as web page http://netweb.usc.edu/\%7ersinha/pkt-sizes/. [Online]. Available: http://www.isi.edu/\%7ejohnh/PAPERS/Sinha07a.html

[25] T. Subramanya and R. Riggio, "Centralized and federated learning for predictive vnf autoscaling in multi-domain 5g networks and beyond," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 63–78, 2021.

[26] I. Martínez-Casanueva, L. Bellido, C. Lentisco, and D. Fernández, "An initial approach to a multi-access edge computing reference architecture implementation using kubernetes," *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, vol. 355, pp. 185–193, 2021.

[27] S. Chaudhry, A. Palade, A. Kazmi, and S. Clarke, "Improved qos at the edge using serverless computing to deploy virtual network functions," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 10 673–10 683, 2020.

[28] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards delay-aware container-based service function chaining in fog computing," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–9.

[29] M. Scazzariello, L. Ariemma, G. D. Battista, and M. Patrignani, "Megalos: A scalable architecture for the virtualization of network scenarios," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–7.

[30] F. Ghirardini, A. Samir, I. Fronza, and C. Pahl, "Model-driven simulation for performance engineering of kubernetes-style cloud cluster architectures," *Communications in Computer and Information Science*, vol. 1115, pp. 7–20, 2020.

[31] M. Gokan Khan, J. Taheri, A. Al-dulaimy, and A. Kassler, "Perfsim: A performance simulator for cloud native microservice chains," *IEEE Transactions on Cloud Computing*, 2021.

[32] The 1st International Workshop on Technologies for Network Twins (TNT 2022), co-located with the 2022 edition of IEEE/IFIP Network Operations and Management Symposium (NOMS). [Online]. Available: https://noms2022.ieee-noms.org/ws4-1st-international-workshop-technologies-network-twins-tnt-2022

[33] Z. Zhang, Y. Zeng, H. Liu, C. Zhao, F. Wang, and Y. Chen, "Smart DC: An AI and Digital Twin-based Energy-Saving Solution for Data Centers," in *Proceedings of 1st International Workshop on Technologies for Network Twins (TNT 2022)*, 2022.

[34] C. von Lengerke, A. Hefele, J. Cabrera, and F. Fitzek, "Stopping the data flood: Post-shannon traffic reduction in digital-twins applications," in *Proceedings of 1st International Workshop on Technologies for Network Twins (TNT 2022)*, 2022.

[35] F. Poltronieri, M. Tortonesi, and C. Stefanelli, "A chaos engineering approach for improving the resiliency of it services configurations," in *Proceedings of 1st International Workshop on Technologies for Network Twins (TNT 2022)*, 2022.

[36] M. Polverini, I. Germini, A. Cianfrani, F. G. Lavacca, and M. Listanti, "A digital twin based framework to enable "what-if" analysis in bgp optimization," in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–6.

[37] P. Krämer, P. Diederich, C. Krämer, R. Pries, W. Kellerer, and A. Blenk, "D2a: Operating a service function chain platform with data-driven scheduling policies," *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 2839–2853, 2022.

[38] M. Groshev, C. Guimarães, A. De La Oliva, and R. Gazda, "Dissecting the impact of information and communication technologies on digital twins as a service," *IEEE Access*, vol. 9, pp. 102 862–102 876, 2021.

[39] F. Laamarti, H. F. Badawi, Y. Ding, F. Arafsha, B. Hafidh, and A. E. Saddik, "An iso/ieee 11073 standardized digital twin framework for health and well-being in smart cities," *IEEE Access*, vol. 8, pp. 105 950–105 961, 2020.

[40] S. H. Khajavi, N. H. Motlagh, A. Jaribion, L. C. Werner, and J. Holmström, "Digital twin: Vision, benefits, boundaries, and creation for buildings," *IEEE Access*, vol. 7, pp. 147 406–147 419, 2019.

[41] Y. Liu, L. Zhang, Y. Yang, L. Zhou, L. Ren, F. Wang, R. Liu, Z. Pang, and M. J. Deen, "A novel cloud-based framework for the elderly healthcare services using digital twin," *IEEE Access*, vol. 7, pp. 49 088–49 101, 2019.

[42] K. M. Alam and A. El Saddik, "C2ps: A digital twin architecture reference model for the cloud-based cyber-physical systems," *IEEE Access*, vol. 5, pp. 2050–2062, 2017.

[43] Q. Qi and F. Tao, "A smart manufacturing service system based on edge computing, fog computing, and cloud computing," *IEEE Access*, vol. 7, pp. 86 769–86 777, 2019.

Davide Borsatti (Member, IEEE) received his B.S., M.S., and Ph.D. in Electronics, Telecommunications, and Information Technologies Engineering from the University of Bologna, Italy, in 2016, 2018, and 2022, respectively. He is currently an Assistant Professor in the Department of Electrical, Electronic, and Information Engineering "Guglielmo Marconi" at the University of Bologna. His research interests include NFV, SDN, intent-based networking, MEC, and 5G network slicing.

Walter Cerroni (Senior Member, IEEE) is an Associate Professor of communication networks with the University of Bologna, Italy. He coauthored more than 150 articles published in the most renowned international journals, magazines, and conference proceedings. His recent research interests focused on multiple aspects of control, management and orchestration of communication network infrastructures, including software-defined networking, network function virtualization, multi-access edge computing, fog computing, service function chaining, intent-based networking systems. He serves/served as Series Editor for the IEEE Communications Magazine, Associate Editor for the IEEE Communications Letters, and Technical Program Co-Chair for IEEE-sponsored international workshops and conferences.

Luca Foschini (Senior Member) received the Ph.D. degree in computer science engineering from the University of Bologna, Italy, in 2007. He is currently an Associate Professor of computer science engineering with the University of Bologna. His interests span from integrated management of distributed systems and services to mobile crowd-sourcing/-sensing, from infrastructures for Industry 4.0 to fog/edge cloud systems. Finally, he is serving as Chair of the ComSoc CSIM TC, and he served in 2022-2023 as Director for the IEEE ComSoc EMEA and non-voting member of the ComSoc Board of Governors.

Genady Ya. Grabarnik (Senior Memeber) is a Professor at Math and CS Department, St John's University. He is a trained mathematician and authored over 120 papers. He spent 10 years at IBM T.J.Watson Research Center where his work was celebrated with a number of awards including Outstanding Technical Achievement Award and Research Achievement Awards. He is a prolific inventor with over 65 US patents. His interests include research in functional analysis, inventions, and research in computer science and artificial intelligence.

Lorenzo Manca is a research fellow at the Interdepartmental Center for Industrial ICT Research of the University of Bologna, Italy. He obtained his master's degree in Telecommunications Engineering from the same university in 2023. His research activity is focused on the application of artificial intelligence (AI) in the domains of service mesh and communication networks.

Filippo Poltronieri (Member, IEEE) received the Ph.D. degree from the University of Ferrara, Italy, in 2021. He is currently an Assistant Professor with the Department of Engineering, University of Ferrara. His research interests include distributed systems, optimization techniques for network and service management, edge and cloud computing, and tactical networks. He has been visiting the Florida Institute for Human & Machine Cognition (IHMC) in Pensacola, FL (USA) in 2016-2017 and 2018.

Domenico Scotece (Member, IEEE) is a junior assistant professor at the University of Bologna, Italy, right after having obtained the Ph.D degree at the same University in April 2020. He received the Master Degree in Computer Science Engineering from the University of Bologna, in 2014. His research interests include pervasive computing, middleware for fog and edge computing, the Software-Defined Networking, the Internet of Things, and 5G network planning and design.

Larisa Shwartz (Member, IEEE) received the PhD degree in mathematics from UNISA University. She is currently DE at the IBM T.J. Watson Research Center, Yorktown Heights, NY. She has research experience in mathematics and computer science, but is now focusing on IT service management technologies for service delivery. She has more than 80 publications and 52 patents.

Cesare Stefanelli (Member, IEEE) received the Ph.D. degree in computer science engineering from the University of Bologna, Italy, in 1996. He is currently a Full Professor of distributed systems with the Engineering Department, University of Ferrara, Italy. At the University of Ferrara he coordinates a Technopole Laboratory dealing with industrial research and technology transfer. He holds several patents, and coordinates industrial research projects carried on in collaboration with several companies. His research interests include distributed and mobile computing in wireless and ad hoc networks, network and systems management, and network security.

Mauro Tortonesi (Member, IEEE) is an Associate Professor and the head of the Big Data and Compute Continuum research laboratory at the University of Ferrara, Italy. He received the Ph.D. degree in computer engineering from the University of Ferrara, in 2006. He was a Visiting Scientist with the Florida Institute for Human & Machine Cognition (IHMC), Pensacola, FL, USA, from 2004 to 2005 and with the United States Army Research Laboratory, Adelphi, MD, USA, in 2015. He participates / has participated with several roles in a wide number of research projects in the distributed systems area, with particular reference to Compute Continuum, Big Data, and IoT solutions in industrial and military environments. He has co-authored over 100 publications and has 4 international patents.

Mattia Zaccarini (Student Member, IEEE) is a Ph.D. student at the Engineering Department of the University of Ferrara. He obtained his Bachelor's degree in Electronics and Computer Science Engineering in 2018 and his Master's degree in Computer and Automation Engineering in 2022 from the University of Ferrara. He is currently part of the Big Data and Compute Continuum research laboratory and his research activity is focused on Compute Continuum, Network Digital Twin, Reinforcement Learning and Computational Intelligence techniques.