OptiFog: A Framework to Optimize the Placement of Microservices in Fog Scenarios

OptiFog: A Framework to Optimize the Placement of Microservices in Fog Scenarios

Claudia Canali, *Member, IEEE*, Giuseppe Di Modica, *Member, IEEE*, Francesco Faenza, *Member, IEEE*, Luca Foschini, *Senior Member, IEEE*, Riccardo Lancellotti, *Member, IEEE*, Domenico Scotece, *Member, IEEE*

***Abstract*—The Fog computing paradigm makes use of dispersed, diverse, and resource-limited devices located at the network edge to effectively implement Internet of Things (IoT) application services that demand low latency and substantial bandwidth. At the same time, the adoption of microservice-based architectures in the IoT domain is on the rise due to their ability to align with the swift evolution and deployment demands of highly dynamic IoT applications and to elastically scale to fulfill load demands. In complex environments like Fog federations, characterized by highly heterogeneous computing and networking resources, the effective allocation of microservices to available nodes, while ensuring compliance with required Quality of Service (QoS) constraints, represents a significant challenge. In this paper, we present the design and implementation of OptiFog, a comprehensive framework that enables users to model, simulate, and validate microservice placement solutions within a realistic testbed environment. Compared to state-of-the-art approaches, OptiFog offers developers a controlled environment for experimenting with placement solutions while providing the assurance that the resulting deployments will meet the targeted QoS requirements in real-world scenarios, specifically in terms of service execution time and energy consumption of Fog nodes. To demonstrate the feasibility of the proposed approach, we implemented and evaluated a representative use case, involving both sub-optimal and optimal microservice placement, and utilizing real-world microservices drawn from the IoT domain.**

***Index Terms*—Microservices placement, Fog computing, Genetic algorithms, Framework, Performance evaluation, Fog federation, Software platform**

I. Introduction

FOG computing is taking place as the reference paradigm to provide latency-sensitive and bandwidth-demanding applications with a distributed computing environment that can satisfy their requirements. Using the presence of Fog nodes, located near end users and IoT devices, and providing computational power, storage capabilities, and networking, Fog computing infrastructures conveniently extend traditional cloud services at the edge of the network [1]. In Fog computing environments, the need to optimally distribute request loads across the available resources provided by Fog nodes is a widely recognized challenge. This problem is further exacerbated in a Fog federation [2], that is a decentralized architectural model in which multiple autonomous Fog computing domains collaborate by sharing heterogeneous computational, storage, and networking resources across administrative boundaries. Although this collaboration enables improved scalability, resource efficiency, and service availability, it simultaneously increases the complexity of the resource allocation problem.

In this domain, most of the literature addresses issues related to optimal placement of services composed of microservice chains, with optimization typically guided by metrics such as service latency and the energy consumption resulting from the service's utilization of computational resources [3], [4], [5]. owever, most of these studies do not assess whether the proposed solutions would actually meet the targeted QoS (which is an input to the problem itself) in real world Fog settings. The proposal discussed in this paper addresses the problem of optimizing microservice placement in a Fog federation environment from both a theoretical and a practical perspective. Researchers usually validate their optimization models through either simulation, using standard or custom-developed simulators [6], [7], [8], [9], or emulation, which entails deploying both the model and the microservices as software prototypes on small-scale, real-world testbeds [10], [11], [12]. *Simulation* enables the rapid exploration of alternative design solutions and supports experimental evaluation in medium- to large-scale environments. Nonetheless, it often falls short in accurately representing all behaviors of real systems, particularly those stemming from hardware–software interactions that are difficult to model in early phases, such as CPU throttling or resource overhead introduced by lightweight virtualization layers. *Emulation*, by contrast, captures many of the dynamics characteristic of real deployments, but offers limited flexibility for exploring alternative designs and generally entails substantially higher implementation costs. For these reasons, we advocate a combined methodology: simulation is first employed to iteratively and cost-effectively refine a candidate solution, after which emulation is used to provide feedback on real-world dynamics that may influence the accuracy and practical feasibility of the proposed approach.

Accordingly, this paper presents OptiFog, a comprehensive framework that supports the optimization of microservice-based service placement problems, enables validation of the computed solutions through simulation, and facilitates robustness testing via deployment of the devised solutions on a small-scale Fog computing emulation environments. The framework is designed as a controlled experimentation environment in which users can develop custom placement optimization strategies, iteratively refine them through a simulation tool, and assess the confidence level with which the resulting

C. Canali, F. Faenza, and R. Lancellotti are with the Department of Engineering "Enzo Ferrari", University of Modena and Reggio Emilia, Modena, Italy. E-mail: {firstname.lastname}@unimore.it.

G. Di Modica, L. Foschini, and D. Scotece are with the Department of Engineering and Computer Science, University of Bologna, Bologna, Italy. E-mail: {firstname.lastname}@unibo.it

Manuscript received April 19, 2005; revised August 26, 2015.

solutions are expected to meet the target QoS requirements in real-world deployment scenarios. The set of features provided by the framework is further extended with a *Microservice Profiler*, a specialized benchmark tool that analyzes the runtime behavior of microservices in an isolated environment and under varying workload conditions, with respect to the QoS metrics that inform placement optimization strategies. Unlike existing frameworks [7], [8], [12], [9], [10], which typically provide only partial support, focusing either on placement policy design or on simulation, or on prototype-based execution, OptiFog offers an integrated workflow that bridges the gap between placement decisions and their actual deployment performance measurement in real Fog environments, enabling a coherent end-to-end assessment of whether the computed placement can reliably satisfy the targeted QoS constraints in deployment.

Following the design principles of the framework, we developed a software prototype of the OptiFog platform and made it publicly available for community use, as detailed in Section V. Furthermore, to demonstrate the practical advantages offered by the proposed framework, we implemented a microservice optimal placement use case. Firstly, we devised an analytical model for a microservice placement problem that accounts for QoS parameters such as service execution time and energy consumption of Fog nodes. Secondly, we present an example of an heuristic that seeks for a placement solution based on genetic algorithms. Through experiments, we demonstrate that the proposed implementation can find a solution in a time that is compatible with the dynamics of the Fog environment. In addition, we assembled a set of services by integrating publicly available microservices sourced from the IoT domain and formulated a microservices placement problem by submitting it to OptiFog, which supported the identification of the placement solution and enabled a two-fold evaluation process that ultimately demonstrated solution consistency within a Fog computing environment. Finally, to highlight OptiFog's flexibility in accommodating different optimization algorithms, we implement a second mechanism to solve the placement problem as a wrapper around a commercial solver to compute an optimal solution for the previously defined scenario. It is worth to note that, unlike heuristics such as the genetic algorithms, this solution does not scale beyond the small scale problem instances tested with our prototypes, but can provide an optimal bound.

Let us conclude by summarizing the main research contributions of this paper, that are as follows:

- The design and prototype implementation of OptiFog, an open-source and ready-to-use software platform that provides the user with tools and services to i) solve microservice placement problems in Fog environments, ii) simulate the solution found, and iii) assess the solution consistency in a small-scale Fog testbed.
- A benchmark tool to characterize microservices' runtime behaviors in terms of expected execution performance.
- A mathematical model to represent a multi-objective microservice placement problem in a Fog scenario.
- A problem solver that resorts to a robust heuristic approach to provide sub-optimal solutions.

Regarding the structure of the article, Section II discusses some related work. The OptiFog design and component architecture are given in Section III. Section IV describes the placement problem and the genetic algorithm- based solution, while in Section V we disclose details of the OptiFog components. Section VI outlines the experimental setup and reports the results, while Section VII offers a discussion of the results and presents the concluding remarks.

II. Related work

In this section, without pretense of being exhaustive, we present the main research areas related to our work by introducing some main efforts therein.

Several research studies in the field of Fog Computing mainly focus on the optimization aspect of the service placement problem [3], [4], [5], aiming at finding the best resource-service pairing that meets multiple criteria, such as the attainment of the QoS demanded by service owners and the need to save on the infrastructure energy consumption. Our reference scenario considers, on one hand, the resources offered by Fog providers in the form of computing nodes and, on the other hand, microservices-powered services, which are a type of service consisting of chains of software modules [14] that inherently offer flexible deployment alternatives. Among the research efforts devoted to optimize microservice placement in Fog systems, Deng et al. [15] formulates the placement problem as an Integer Linear Programming (ILP) model with the objective of minimizing the cost of application deployment while adhering to resource constraints and ensuring the expected service response time. The proposed placement algorithm is designed to handle the deployment of a single application at each iteration. However, the microservice placement problem belongs to the class of NP-hard problems, which implies that exact optimization solutions are impractical for large-scale instances.

To effectively manage the complexity of multi-objective placement problems, researchers often rely on sub-optimal solutions based on heuristics, frequently leveraging bio-inspired or evolutionary approaches [5], [16], [17], with Particle Swarm Optimization (PSO) and Genetic Algorithms (GA) being the preferred approaches [6], [18], [19], [20]. In [6], authors propose a solution based on PSO aimed at maximising the satisfaction of multiple QoS parameters in a context with limited Fog resources; the solution is evaluated in a simulated Fog environment. Guerrero et al. [18] compare three evolutionary algorithms, Weighted Sum Genetic Algorithm (WSGA), Non-dominated Sorting Genetic Algorithm (NSGA-II) and multi-objective evolutionary algorithm based on decomposition (MOEA/D), for solving Fog service placement focusing on optimising latency, service spread and use of resources. In [19], an improved GA is proposed for microservice placement with the goal of costs minimization, while in [20] GAs are exploited to optimize task execution by distributing workloads across edge devices and potentially the cloud, considering both energy consumption and task completion time. In our framework, we propose a heuristic based on a non-standard GA, specifically developed to fit our placement

TABLE I: Synthetic view of state-of-the-art frameworks for microservices placement

	Placement Policy Integration	Simulator	Service Profiler	Service Orchestrator	Emulator
MLE [7]	-	✓	-	✓	-
FogPlan [8]	-	✓	✓	✓	-
PTC [12]	-	-	✓	-	✓
FogBus2 [9]	✓	✓	-	✓	-
MicroFog [10]	✓	-	-	✓	✓
Con-Pi [13]	✓	-	-	-	✓
OptiFog	✓	✓	✓	✓	✓

problem, whose preliminary version was presented in [21]. Our modified GA leverages an innovative service chain model, taking into account that typically not all microservices are invoked within all workflow instances and that fog nodes can be switched off to reduce energy consumption. The main modifications involve the representation of the GA chromosome and the mutation and crossover operators, as described in Section IV-G. Other studies exploit predictive approaches for service placement based on Deep Reinforcement Learning (DRL) [22], [23], [24] in order to react to fast-changing Fog environments. However, service placements based on DRL are typically characterized by long learning times to converge and need a high volume of errors to explore. With regard to the proposed OptiFog framework, a distinguishing feature lies in its modular design, which enables the seamless integration of diverse service placement strategies. This is achieved through the implementation of the problem solver component as a microservice exposed through a lightweight RESTful interface (see Section V). This design would also support the integration of reinforcement learning mechanisms to further refine and enhance placement decisions.

Focusing on frameworks for microservices-based applications development in Fog environments, several works have been recently proposed in the literature. We summarise the main differences between OptiFog and related state-of-the-art frameworks in Table I. To support a consistent interpretation of the comparison reported in the table, we briefly clarify the meaning of the criteria adopted. *Placement Policy Integration* denotes the ability of a framework to host different placement strategies through modular and interchangeable components, rather than enforcing a fixed or embedded policy. *Simulator* refers to the presence of a simulation engine to evaluate the solutions, while *Service Profiler* indicates a component capable of empirically characterizing the runtime behavior of microservices as opposed to relying solely on synthetic or pre-defined service models. *Service Orchestrator* identifies the capability to automatically deploy, start, and manage microservices on distributed Fog nodes according to a placement configuration. *Emulator* indicates a testbed or prototype environment where microservices and networking conditions are instantiated on real or virtualized resources to assess deployment robustness under realistic operating conditions.

In [7], authors propose the MLE framework for microservice placement, instance scaling and request execution for IIoT applications. The framework exploits an ad-hoc algorithm for the service placement, and the performance are evaluated through simulation. FogPlan, a framework consisting of a centralised Fog Service Controller responsible for hosting the data stores, provisioning Fog services and deploying them within Fog nodes, is presented in [8]. In [12] authors present PTC, a microservice deployment and execution model that uses a reinforcement learning mechanism to decide the target fog node for an application microservice based on its prior observation of the system's states; performance are evaluated in a testbed setup. All these solutions support deployment and composition of microservices, but do not offer the capability to integrate different placement and load balancing policies. Deng. et al. [9] proposed FogBus2, a resource management framework for the deployment of containerized applications across Fog and Cloud resources. In [10], authors present a Fog framework compatible with cloud native technologies, providing a control engine that executes placement algorithms and deploys applications across federated Fog environments. Differently from these frameworks, Con-Pi [13] is a framework based on a centralized controller to execute integrated placement policies and deploy containerized microservices that, like our proposal, includes in the optimized placement management not only performance-oriented objectives, but also the goal of minimizing the overall energy consumption of the Fog nodes.

To evaluate the performance of the proposed frameworks, cited studies exploit simulation [7], [8], [9] or alternatively small-size prototypes of federated Fog or Fog–Cloud environments [12], [10], [13]. As for, our proposal it aims at providing Fog federation owners with a platform offering both simulation and emulation services, in order to evaluate the confidence with which the expected performance will be met in a real-world deployment conditions. Moreover, most of the cited studies lack a dedicated service profiling capability and rely on synthetic or static service characterizations. These abstractions fail to capture essential aspects of real microservice behavior, such as variance in execution times under different loads: placement decisions taken on the basis of such simplified characterizations may satisfy QoS constraints in simulation while failing to do so in practice. In our proposal, we claim for the integration of a service profiler in the framework that allows OptiFog to base placement decisions on empirically observed behavior. For these characteristics, our work goes beyond the current state of the art in that it proposes a framework that fog infrastructure administrators and developers can use to estimate whether a given placement scheme is actually capable of attaining the expected performance in terms of QoS and energy consumption. The design of the OptiFog framework is thoroughly discussed in the following section.

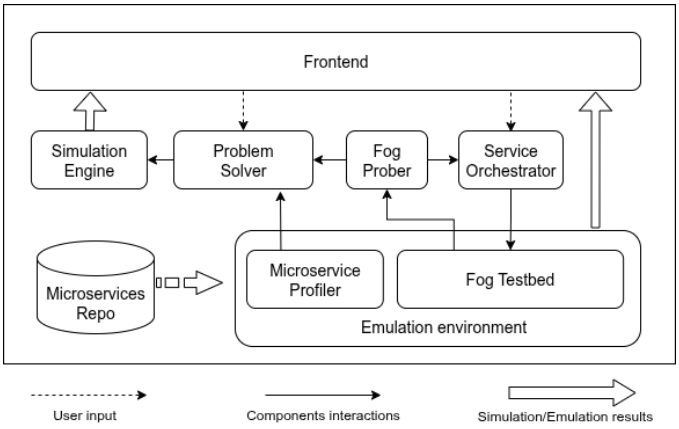


Fig. 1: The OptiFog Platform architecture

III. Framework design

OptiFog is designed to help users develop optimal microservice placement on computing nodes provided by a Fog Federation, that is, a Fog system comprising a number of geographically sparse *Fog domains* forming a federation of Fog nodes. OptiFog offers a comprehensive suite of tools for addressing microservice placement problems and evaluating the robustness of the resulting solutions. Users are expected to formulate a placement problem and submit it to the platform, which offers: i) a software tool for analytically solving the placement problem, ii) a simulation environment to evaluate the effectiveness and robustness of the computed solution, iii) a testbed environment to estimate performance under small-scale real-world conditions, and iv) an orchestrator for deploying services in a production setting. In particular, the simulation and emulation components play a crucial role in demonstrating that the placement computed by the analytical solver can be expected to reliably meet the target QoS requirements when deployed in a real production environment.

In Figure 1, we show the component-based architecture of the platform along with the interaction of the main components. The architecture includes the following entities:

- *Microservice Repository*. It is a pool of microservice images that the service chains are composed of. The repository is meant to serve the Microservice Profiler for benchmarking purpose. Furthermore, it is used by the Orchestrator to retrieve the microservice images that need to be rolled out according to the placement plan.
- *Fog Prober*. Together with the Microservice Profiler, the Fog Prober constitutes a key component that supports the accurate modeling of the service time of a microservice chain. It is in charge of sensing the Fog execution environment and providing robust measures of the parameters that most impact the service performance. In this work, we will account for parameters such as the nodes' available computing power and the node-to-node network delay. The Problem Solver will use the data received from the Fog Prober to decide whether a new microservice placement scheme must be enforced to sustain the QoS levels. Furthermore, the gathered data will inform the Service Orchestrator to promptly take fine-grained counteractions (e.g., re-spawn of a failing container).
- *Microservice Profiler*. This component is responsible for fetching a microservice image, running an instance of it in a local sandbox, probing it with a synthetic requests traffic, and eventually producing a microservice "profile". This component adopts a close-box approach to the problem of predicting the performance of microservices once they are rolled out in a real production environment. This information is essential for the Problem Solver to accurately model the response time of a request sent through a microservice chain. In this work, we focus on the characterization of a microservice in terms of its response time and energy consumption. The reader will find more details on this component in Section V-B.
- *Problem Solver*. When fed with the placement problem, measurements of the Fog context provided by the Fog Prober, and microservices profiles elaborated by the Microservice Profiler, this component searches for a solution to the placement problem. We delve into the technical details of the Problem Solver in Section V-A.
- *Simulation Engine*. Following a simulation-based approach, this tool enables the Solution Developer to address the placement problem in a cost-effective manner. In particular, it provides a software-simulated environment through which the developer can iteratively refine the problem parameters and evaluate the resulting solution until it satisfies the expected service-level requirements. Details on this component are disclosed in Section V-C.
- *Fog testbed*. This component offers a software environment emulating a small-scale fog which the *Software Developer* can leverage for assessing the performance of the placement solution produced by the Problem Solver and simulated via the Simulation Engine. Its purpose is to estimate the potential performance of the proposed solution under realistic conditions, thus anticipating how it would behave in a full-scale production system. Details about the Fog Testbed can be found in Section V-E.
- *Service Orchestrator*. Within the OptiFog framework, the orchestrator is responsible for managing the deployment, configuration, and execution of microservices in the Fog testbed, in accordance with the specifications defined by the placement solution. It automates the operations required for Service Developers to retrieve microservice images and deploy containerized services in an efficient and streamlined manner.
- *Frontend*. This component enables end users to interact with OptiFog. End users are allowed to submit placement problems, instruct the deployment of microservices in the Fog Testbed, collect and visualize results of both simulations and emulations.

In Fig. 2, we depict a typical usage scenario of the OptiFog platform. Let us assume that a set of composite applications are demanding a share of Fog resources that can guarantee the provisioning of certain QoS levels. We also assume that each application requesting the assignment of computing resources

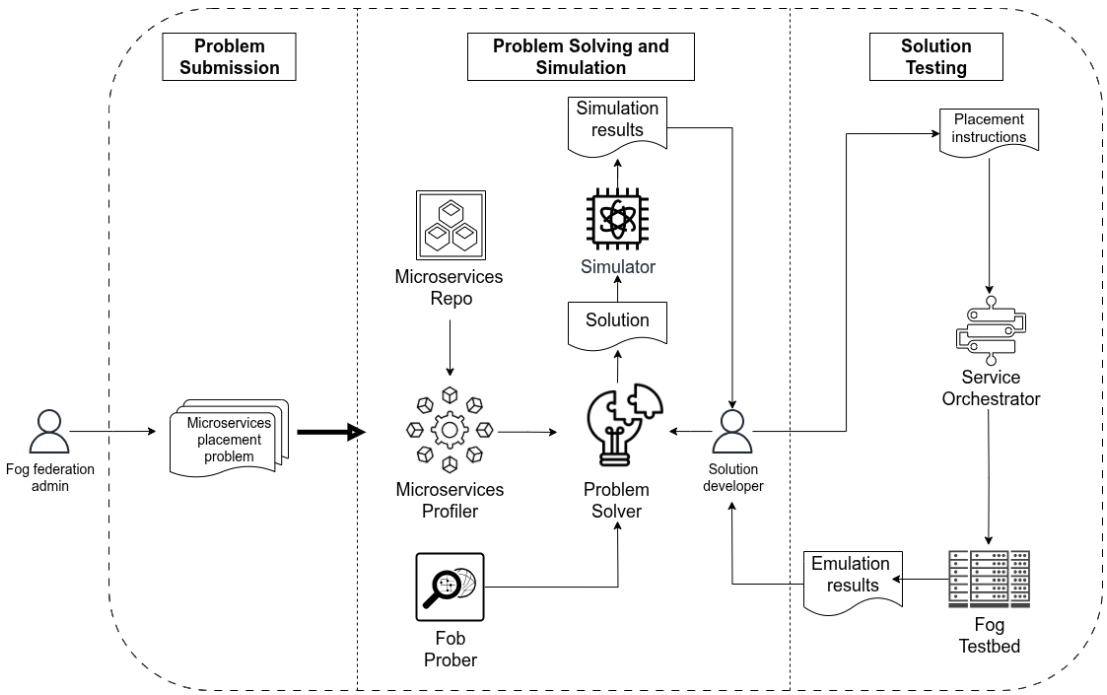


Fig. 2: The OptiFog usage scenario

is composed of microservices that collaborate with one another in a distributed fashion. For simplicity, in the remainder of the paper, we will refer to the set of microservices that make up an application as a *service chain*. In addition, we will use the terms *application* and *service* interchangeably.

We define the placement problem as a typical optimization problem, where the input includes a set of service chains, each composed of a limited number of microservices, the projected request load for each service chain, and the QoS constraints imposed by end-user requirements. In this study, QoS is defined using two metrics: the average service response time, measured from the moment a request enters the service chain to the point at which a response is delivered, and the energy consumed by the service while executing. The *Fog federation administrator* is responsible for formulating and submitting a placement problem to the platform, as well as enforcing the resulting placement solution across the Fog federation to ensure that the QoS requirements of each service chain are satisfied. The *Solution developer* is tasked with developing, running, and validating a microservice placement solution. As illustrated in Figure 2, upon submission of a placement problem, the Fog Prober and the Microservice Profiler components are activated to probe the Fog Testbed and construct the performance profiles of all microservices involved, respectively. In particular, the former is tasked with monitoring Fog system parameters such as round-trip time (RTT) between Fog nodes, as well as the nodes' CPU and RAM utilization, while the latter handles the retrieval of microservice images from the repository and performs benchmarking in an isolated environment to extract microservice performance profiles. The information obtained in this manner is provided to the Problem Solver component, which then computes a solution to the placement problem. The proposed solution can subsequently be evaluated using the Simulation Engine and, later, tested on the Fog Testbed. To deploy the microservices on the Fog testbed in accordance with the provided placement solution, the *Service Developer* invokes the Service Orchestrator, supplying it with the corresponding deployment instructions.

The *Solution Developer* has access to the results produced by both the Simulation Engine and the Fog Testbed, and oversees the entire solution validation process. The proposed two-step validation process, which combines simulation and emulation, offers advantages that neither approach can guaranty when used in isolation. Moreover, our approach attempts to mitigate the typical limitations of simulation, particularly the lack of precise system-level data, by introducing the Fog Prober to extract realistic operational parameters (e.g., CPU and RAM utilization of each Fog node, RTT between nodes) directly from the Fog Testbed, and by leveraging the Microservice Profiler to obtain realistic traces of microservice behavior under production conditions.

Eventually, a close alignment between simulation and emulation outcomes provides assurance that the devised solution can effectively satisfy the QoS requirements under real production conditions. In the presence of an outcome mismatch, the validation process nevertheless yields a meaningful estimation of the discrepancy between the simulated and experimental results.

IV. Microservice Placement Problem

In a Fog context, the *microservice placement problem* is the problem of deciding which of the computing nodes available in the Fog infrastructure are fit to guarantee the overall QoS demanded by a pool of microservice-based applications. There are several aspects that make the problem hard to

solve, such as, e.g., the number and resource greediness of microservices, the heterogeneity of available Fog nodes, the workload generated by end users, etc. This section provides a formal definition of the microservice placement problem, detailing the analytical modeling of the main components of the Fog system and the formulation of the associated global optimization objective. Finally, we present a heuristic based on genetic algorithms to solve the placement problem with execution times compatible with the dynamics of the target environment.

A. Model overview

We define the microservice placement problem as a performance problem with the goal of minimizing energy consumption as well as the overall response time of multiple service chains co-located in the same Fog infrastructure.

In our model, we assume a model where requests visit the nodes where microservices are deployed according to the service chain definition. The requests must wait in the ready process queue until the Fog node CPU is available. The model does not explicitly consider parallel execution of requests (either due to parallel requests to a service or request to different but co-located services) and uses a simplified approach of considering different processing speed of the requests to capture the typically heterogeneous nature of Fog infrastructures. A multi-processor performance model can be introduced in the queuing theory mathematical modeling at the price of a more complex formulation and longer execution times of the problem solver and of the simulation, but this is left as a possible extension of the present work. The response time of a generic service chain depends on the execution time of the microservices composing the chain, on the queuing time on the Fog nodes where the services are deployed and on the delays data packets experience through the network, as data are may flow from node to node. Furthermore, we assume that each service chain is subject to a Quality of Service constraint in the form of a maximum average response time that must be granted.

In our model, we define service chains as applications composed by a sequence of microservices. Each microservice can be placed on a Fog node. Throughout the paper, we refer to Table II as a summary for the notation used in the model.

B. Microservice performance model

We assume that a generic service chain $c \in \mathcal{C}$ is composed of a set of microservices. With λ_c we refer to the incoming request load to the service chain.

A microservice $m \in \mathcal{M}_c \subset \mathcal{M}$ belonging to the service chain is characterized by the average service time S_m and the service time's standard deviation σ_m. For a microservice, we also indicate with F_m the probability of not participating to the service chain execution. This is aimed to capture situations such as failed prerequisite validation in the service (for example, if a service aiming to classify anomalous people behaviors in surveillance footage identifies the scene as empty). If the request rate for service chain c is λ_c, we can define the invocation rate λ_m of microservice $m \in \mathcal{M}_c$ as:

TABLE II: Notation and parameters for the proposed model.

Model parameters	
$\mathcal{M}$	Set of microservices
$\mathcal{M}_c$	Set of microservices of chain c
$\mathcal{F}$	Set of Fog nodes
$\mathcal{C}$	Set of service chains
λ_m	Incoming request rate to microservice m
λ_f	Incoming request rate to Fog node f
λ_c	Incoming request rate to service chain c
Λ	Incoming global request rate
S_m	Average service time for microservice m
σ_m	Standard deviation of S_m
F_m	Chain exit probability at microservice m
P_f	Computational power of Fog node f
W_f	Average waiting time on Fog node f
S_f	Average service time on Fog node f
σ_f	Standard deviation of S_f
R_c	Average response time for service chain c
T_c^{QoS}	QoS requirement of service chain c
$P_{min,f}$	Idle power consumption of Fog node f
$P_{max,f}$	Maximum power consumption of Fog node f
o_{m_1,m_2}	order of execution of microservices in a chain
δ_{f_1,f_2}	network delay between nodes f_1 and f_2
Model indices	
f	A Fog node
c	A service chain
m	A microservice
Decision variables	
$x_{m,f}$	Allocation of microservice m to Fog f
e_f	set to 1 if Fog node f is used

$$\lambda_m = \lambda_c \prod_{\substack{m' \in \mathcal{M}_c \\ m' \prec m}} F_{m'}$$

C. Fog node performance model

Let us now focus on a single Fog node. Each node may host several microservices and can be either switched on or off. We start focusing on a generic Fog node f that is turned on (that is, $e_f = 1$).

Let $x_{m,f}$ be a Boolean decision such that $x_{m,f} = 1 \iff$ service m is hosted on Fog node f. The resulting service time on Fog node f is a mixture of distributions related to the hosted microservices. The average S_f service time of such distribution and its variance σ_f^2 are defined as:

$$S_f = \frac{1}{P_f} \cdot \sum_{m \in \mathcal{M}} x_{m,f} \frac{\lambda_m}{\lambda_f} S_m \tag{1}$$

$$\sigma_f^2 = \left(\frac{1}{P_f^2} \cdot \sum_{m \in \mathcal{M}} x_{m,f} \frac{\lambda_m}{\lambda_f} (S_m^2 + \sigma_m^2) \right) - S_f^2 \tag{2}$$

where λ_f is the arrival request rate on node f and is defined as: $\lambda_f = \sum_{m \in \mathcal{M}} x_{m,f} \lambda_m$. The terms S_m and σ_m represent microservice m's average service time and standard deviation respectively. We highlight that these parameters constitute the microservice's profile and are provided by the Microservice

Profiler upon completion of its benchmarking process, as discussed in Section III.

If we consider a Fog node as a queuing server, with a Poisson arrival process, the waiting time W_f on Fog node f can be described using the Pollaczek-Khinchin equation:

$$W_f = \frac{S_f^2 + \sigma_f^2}{2} \cdot \frac{\lambda_f}{1 - \lambda_f S_f} \tag{3}$$

The assumption behind Eq. (3) is that the arrival rate to every Fog node is a Poisson process so that the overall queuing system can be expressed in product form. This assumption is not always valid: a service time with non-exponential distribution determines an inter-arrival time that is not a Poisson process for the subsequent nodes. Experiments carried out with simulation [25] suggest that queuing theory can still be applied with limited impact on the overall model precision as long as (1) many microservices are located on a Fog node or (2) the standard deviation of the service time of the microservices is close to the mean value. As long as the microservices placement satisfied the requirements, we can still rely on Eq. (3) to approximate the waiting time on a Fog node.

On the other hand, a turned-off node f is marked by the decision variable $e_f = 0$ and cannot host any microservice.

D. Service chain performance model

We can define the response time of a service chain c as the sum of three components: the sum of the service times of the microservices $m \in \mathcal{M}_c$; the waiting time on the Fog nodes where the microservices are placed; the network delays due to the inter-Fog communications.

$$R_c = \sum_{\substack{m \in \mathcal{M}_c \\ f \in \mathcal{F}}} x_{m,f} \left(W_f + \frac{S_m}{P_f} \right) + \\ + \sum_{\substack{m,n \in \mathcal{M}_c \\ f,g \in \mathcal{F}}} o_{m,n} \cdot x_{m,f} \cdot x_{n,g} \cdot \delta_{f,g} \tag{4}$$

where $o_{m,n}$ is a parameter used to identify consecutive microservices in a service chain: $o_{m,n} = 1 \iff m \lessdot n$, meaning that service m is invoked just before n in the service chain.

E. Fog node energy model

Our model assumes that the energy consumed by a Fog node depends only on the node utilization, with a linear function that correlates utilization and power consumption. Energy is defined as the power consumption multiplied by the Fog nodes that are turned on (Δt). We define the function as $E(\cdot)$ as follows:

$$E_f = e_f(P_{min,f} + (P_{max,f} - P_{min,f})\lambda_f S_f)\Delta t \tag{5}$$

If a node is turned off ($e_f = 0$), no power consumption occurs. On the other hand, if a node is turned on, power consumption grows from $P_{min,f}$ to $P_{max,f}$ as the node utilization ($\lambda_f S_f$) increases. The critical parameters $P_{min,f}$ and $P_{max,f}$ can be experimentally obtained using the Emulator provided by the framework. We provide a set of preliminary results used to tune our experimental evaluation in Appendix **??**.

F. Problem formulation

Having provided a model for the Fog system, we can now define an optimization problem for the placement of microservices on the distributed infrastructure.

$$\text{lexmin}(obj_E, obj_R) \tag{6}$$

$$obj_E = \sum_{f \in \mathcal{F}} E_f \tag{7}$$

$$obj_R = \sum_{c \in \mathcal{C}} \frac{\lambda_c}{\Lambda} R_c \tag{8}$$

subject to:

$$\sum_{f \in \mathcal{F}} x_{m,f} = 1 \quad \forall m \in \mathcal{M}, \tag{9}$$

$$\lambda_f S_f \leq e_f \quad \forall f \in \mathcal{F}, \tag{10}$$

$$R_c \leq T_c^{QoS} \quad \forall c \in \mathcal{C}, \tag{11}$$

$$x_{m,f}, e_f = \{0, 1\}, \quad \forall m \in \mathcal{M}, f \in \mathcal{F}, \tag{12}$$

The problem goal described in Eq. (IV-F) aims to lexicographically optimize the two objectives Eq. (7) and (8). Eq. (7) defines the first objective which is the energy consumption defined as the sum of the energy consumed by each Fog node. Eq. (8) is the average response time of the service chains (weighted by their invocation frequency, where λ_c is the invocation frequency of the service chain c and Λ is the incoming global request rate). In lexicographic optimization the second objective is taken into account only as long as the first objective remains minimum. The stepwise nature of the energy objective (where major changes in obj_E occur when Fog nodes are switched on or off), and the constraint in Eq. (11) that deremines $obj_R \leq \max T_c^{QoS} \forall c \in \mathcal{C}$ can be used to rewrite the lexicographic optimization as a linear combination of the two objectives such that $obj = obj_E + \epsilon \cdot obj_R$, where the value ϵ is chosen to ensure that obj_R remains at least two orders of magnitude lower than the step in the energy function $\epsilon \leq 0.01 \min(P_{min,f} \forall fin \mathcal{F})$. This approach ensures that the error in the first and most important energy objective remains below 1%.

The solution space of the optimization problem is bounded by these constraints:

- Eq. (9) forces the allocation of each microservice to exactly one Fog node
- Eq. (10) avoids overload condition for every Fog node ($\lambda_f S_f$ is the load on Fog node f and it should not exceed 1). The presence of the right hand side element e_f means that powered-off nodes must be idle.
- Eq. (11), captures the QoS requirement of each service chain. In our paper we consider $T_c^{QoS} = K \cdot \sum_{m \in \mathcal{M}_c} S_m$, where $K = 10$. This is derived from the literature [26]
- Eq. (12), captures the Boolean nature of the decision variables $x_{m,f}$ and e_f

G. Optimized microservice placement

The microservice place is based on a non-linear optimization problem, due to the non-linear response time and energy consumption objectives.

In order to tackle such types of problems heuristics and meta-heuristics should be used. The possibilities are either to rely on existing solvers that implement general purpose heuristics, such as CPLEX[1], Gurobi[2] or K-Nitro[3]. These solvers can be used as a module of a general-purpose optimization modeling solution such as AMPL[4]. An alternative approach is to rely on a heuristic specifically tailored to the problem, for example relying on evolutionary approaches such as genetic algorithms, already proposed to address similar problems [27], [28], [25]. In the present paper, we explored both solutions to fully demonstrate that the OptiFog framework can easily integrate different approaches and different underlying technologies as long as the interfaces remain the same.

For the sake of clarity, we outline the characteristics of the GA approach developed to solve the placement problem, pointing to a specific publication for the details of the proposed algorithm [21]. The choice of considering genetic algorithms in place of other metaheuristics such as Particle Swarm Optimization or Variable Neighborhood Search is motivated by a comparison service placement algorithms for Fog computing available in literature [29] that suggest that GA provide high-quality solutions, are stable over a side range of problem characteristics, and require a limited metaheuristic parameter tuning.

In genetic algorithms, a solution is encoded as a string of symbols (*chromosome*), with each symbol being a *gene*. A population of *individuals* (represented by chromosomes) is randomly generated and the population evolves through *generations* to identify the best candidate solutions.

In the genetic algorithm developed to solve the microservice allocation problem, the chromosome is composed of several sequences starting with a Fog node identifier and followed by a list of microservices hosted on that Fog node. The length of a chromosome depends on the number of active Fog nodes and ranges from $\|\mathcal{M}\|+1$ (only one Fog node used) to $\|\mathcal{M}\|+\|\mathcal{F}\|$ when all Fog nodes are active. This representation captures the nature of the problem and captures the constraints (9) and (12). It is worth to note that this chromosome encoding is a qualifying point of our GA implementation as it ensures that the nodes available to process requests are passed from a generation to the next. As the population evolves, solutions with fewer nodes that can still meet QoS requirements are more likely to be passed to the next generation. The result is a population that automatically converges to the least possible number of Fog nodes. In case of a constraint violation, the considered solution is not feasible. We maintain these solutions in the genetic pool to avoid being stuck at a local minimum. However, we add a penalty to the objective functions. For the overload avoidance constraint in Eq. (10) the penalty is proportional to $\lambda_f \cdot S_f$ for every overloaded node. For QoS satisfaction constraint in Eq. (11) the penalty is proportional to $R_c - T_c^{QoS}$; if it is not possible to compute R_c (for example in case of overloaded nodes) R_c is set to $10 \cdot T_c^{QoS}$. Penalty weights are chosen based on the genetic pool of each generation. We make sure that obj for the best infeasible solution is one order of magnitude higher than obj for the worst feasible solution. At the same time, we preserve the ordering of infeasible solutions to make the least infeasible ones preferable. This helps the algorithm converge by avoiding being stuck in a local minimum, but we can still prune over the generations solutions that have a higher degree of constraint violation. At the end of the GA evolution, if no feasible solution is present in the genetic pool, the algorithm returns the best solution found, indicating that the solution is infeasible. For the algorithm, we use tournament selection keep in the genetic pool only the fittest individuals, a mutation operator that can swap two genes (migrating a service from one node to another), and a modified ordered crossover operation that can cope with the possibility of having two parents of different length.

The heuristic based on the genetic algorithm is implemented using the DEAP library[5].

V. Platform components

We implemented the OptiFog platform components as RESTful microservices, so that the platform itself can easily be deployed in a distributed and scalable software environment. In the following, some implementation details of the core components are disclosed.

A. Problem solver

A placement problem is submitted to Problem Solver as a POST HTTPS request. The request body is a JSON representation of the problem, with the service chains definition, the description of the Fog nodes (including their computational capacity) and a network delay matrix. The solver produces a solution to the problem. The solution, encoded as a JSON data structure, contains the placement scheme of the microservices on the nodes of the infrastructure. An example of the JSON data structures used in the test problem is provided in Appendix **??**. To demonstrate the potential of the proposed framework, two different implementations of the solver have been developed: one based on the genetic algorithm and another based on the AMPL processing language as the front-end for the K-Nitro solver. The solution can either be provided directly in the body of the HTTP response, or when the problem size determines a response time incompatible with synchronous interactions, a callback can be specified: when the solver reaches a solution, the callback endpoint is invoked and the problem solution is passed using an HTTP POST method. Moreover, the use of RESTful APIs are standard, which allows interchangeable optimization heuristics (e.g., GA vs AMPL/K-Nitro) to be plugged in transparently into the workflow.

[1]https://ampl.com/products/solvers/linear-solvers/cplex/
[2]https://ampl.com/products/solvers/linear-solvers/gurobi
[3]https://ampl.com/products/solvers/nonlinear-solvers/knitro/
[4]https://ampl.com/

[5]DEAP: Distributed Evolutionary Algorithms in Python - https://deap.readthedocs.io/en/master/

B. Microservice Profiler

The Microservice Profiler is tasked with extracting a set of microservice features that are relevant to the placement problem. In the context of this study, the micorservice features of interest are: i) execution time, defined as the duration required for a microservice to respond to a request, and ii) energy consumption, which refers to the CPU energy expended while serving a request. As discussed in Section IV, the service execution time is incorporated into the first objective function of the optimization problem (Eq. 7), while energy consumption is used to formulate the second objective function (Eq. 8). In its current implementation, the Profiler is capable of automatically extracting the execution time of microservices, whereas the measurement of energy consumption remains a manual process, which is planned to be automated in future work.

The extraction of microservice features is conducted within an isolated and instrumented environment deployed on the Fog Testbed. The Profiler is implemented as an independent microservice and exposes a RESTful interface through which profiling instructions for a given microservice can be submitted. The profiling invocation for a microservice includes the following parameters:

- The URL of the target microservice
- The HTTP method to be employed
- The input payload of the request
- The number of requests to be executed during the profiling session

Upon receiving a profiling request, the Profiler initiates a procedure to measure the execution time of the target microservice. Internally, this procedure leverages the Apache Benchmark (ab) tool [30] to issue requests to the microservice. In the current implementation, the ab tool is configured to send requests sequentially, in accordance with the requirements of the placement problem (see Section IV). Future versions of the Profiler are expected to incorporate additional stress testing modes to support more diverse performance evaluation scenarios. Upon completion of the profiling process, the client who submitted the request can retrieve the average response time and the corresponding standard deviation, calculated over the set of executed experiments, via a dedicated API. All profiling modules utilized in this study, along with the complete source code of the Profiler, are publicly available in our GitHub repository [6].

As already stressed in Sec. II, we advocate that while microservice profiling is a widely adopted technique for performance characterization and resource estimation, its integration within Fog computing environments into an automated pipeline where real performance data informs both optimization and validation processes remains limited and insufficiently explored.

C. Simulation Engine

The simulator is a container-based implementation based on the Omnet++ simulation framework. Additional modules, specifically developed for this task, provide a model for the Fog nodes operations and for the routing of requests between Fog nodes, according to the service chain logic[7]. Additional code has been developed to support the automatic execution of multiple runs and the collection of relevant data[8]. The choice of creating new modules for a simulation framework, rather than adapting an existing simulators (such as iFogSim [31]) lies in our need of accurately modeling the response time of every request to fully validate our model assumptions. Most existing simulators for cloud computing environments model applications as resource-requesting entities, without the possibility of specifying the inner mechanisms for request processing.

The simulator accepts as input a JSON-based structure containing:

- The service chains to be deployed, together with the description of the related list of microservices, each characterized by the service time (mean and variance). In the case where a service chain may be stopped before executing every microservice (for example, as a consequence to input validation), the exit probability after every step of the service chain is provided.
- The workload description, with the number of requests per time unit in input to each service chain.
- The description of the Fog infrastructure, with the list of the nodes, their processing power and the matrix of network delays connecting each pair of Fog nodes.
- The deployment scheme, mapping the microservices to the Fog nodes. The deployment scheme is the main output of the optimization heuristic described in Section IV-G.

The simulator models a queuing network-based model where each Fog node is a queue receiving jobs belonging to different services and with a different service time taken from the microservice description. The simulation runs for a pre-defined amount of simulated time (long enough to have a statistically significant amount of data in output from the simulator). For each scenario, multiple runs of the simulator, with different random number generator seeds are performed, in order to provide a confidence interval for the simulator output. The output of the simulator are:

- The average response time of each service chain, together with a breakdown in network delay, service time and queuing time. For every parameter, the standard deviation is also returned.
- The utilization of the Fog nodes, including also the average value and the standard deviation of service time, and waiting time of jobs in every node.

D. Service Orchestrator

The Service Orchestrator is responsible for enforcing the deployment of microservices according to the placement scheme provided by the Problem Solver. To accomplish this task, we leveraged the services offered by the *TORCH* tool[32], an open-source cloud orchestrator that adheres to industrial

[6]https://github.com/rlancellotti/GAFog

[7]https://github.com/rlancellotti/fogchains

[8]https://github.com/rlancellotti/omnet_analyzer

standards such as OASIS TOSCA[33] and OMG BPMN[34], that is capable of automating the provisioning of microservices in Multi-Cloud and Fog environments. The main reason motivating this choice is that TORCH adopts a strategy that separates the provisioning workflow from the actual invocation of container engines API. The merit of such approach is that support to potentially any virtualization infrastructure (both public and private, in the Cloud or in the Fog) can be added at a very low implementation cost. In the context of this work, we equipped TORCH with a connector (i.e., a software driver) that enabled it to interface with the Containernet tool installed on the emulator environment and to deploy dockerized microservices over it. Another relevant feature of TORCH is its compatibility with the TOSCA standard, which offers a standardized way of representing topologies and deployment schemes of Cloud-native applications. TORCH exposes RESTful APIs that the Problem Solver invokes to submit placement instructions which, according to the latest version of the standard, are provided in a very simple YAML-based notation[9].

The TORCH approach enables a very loose coupling between the Service Orchestrator and the Fog Testbed. This design choice allows the TORCH orchestrator to be employed with minimal effort both within the OptiFog framework—for the purpose of validating optimization solutions—and for the final deployment of microservices in a real Fog federation production environment. To this end, it is sufficient to replace the connector interfacing with Containernet with alternative connectors capable of interacting with the virtualization environments (which may differ across domains) present in the various administrative regions that make up the Fog federation. Furthermore, in our future plan the Fog Testbed will evolve towards a Kubernetes-powered runtime. With the new environment, on the service orchestration side the same placement instructions will still be provided in the TOSCA notation, while a new connector will have to be implemented in order for TORCH to correctly invoke the new computing runtime API.

E. Fog Testbed

The Fog Testbed component offers to solution developers a service to emulate the placement solution in a controlled environment that reproduces the features of a real Fog context. At the core of the emulation environment lies Containernet [35], an extension of the well-known Mininet network emulator that supports the deployment of Docker containers as hosts within emulated network topologies.

The component's front-end is a RESTful interface that accepts emulation requests and hands them over to the Containernet. A request embeds a JSON file that represents the configuration of the Fog context to be emulated (an excerpt of the JSON file is provided in Appendix **??**). This configuration file defines the Fog network topology to be emulated, including elements such as switches and nodes. It also specifies the set of microservices to be deployed as Docker containers on virtual computing nodes. For each microservice, configurable parameters such as the Docker image and the allocated CPU share can be explicitly defined. Finally, a section of the JSON file is devoted to the specification of the service workload, i.e., the load of user requests that the services will have to serve when is put in production. To this aim, the emulator too makes use of the *Apache ab* tool [30] to trigger HTTP requests according to a desired pattern.

The Fog Testbed hosts lightweight, containerized software agents responsible for monitoring the status of the Fog system. In the current implementation, these agents are capable of collecting both network-level metrics such as node-to-node round-trip time (RTT), and compute-related metrics, including CPU and RAM utilization of individual nodes. For metric collection, the system leverages well-established native tools available within the virtualized environment. All gathered data are transmitted to the Fog Prober, which will make them accessible to the Problem Solver and to the Service Orchestrator upon request.

F. Fog Prober

This component is responsible for collecting measurements from the Fog Testbed environment. In the current platform prototype, the Fog Prober is deployed as a containerized service exposing two RESTful interfaces: one directed toward the software agents hosted within the Fog Testbed, enabling the collection of both network-level and node-level metrics; and another directed toward the Problem Solver and the Service Orchestrator, which provides access to the collected data. In this study, the Prober was tested only for the provisioning of Fog context information to the Problem Solver, i.e., at solution design time. As part of our future work, we plan to replace the custom Fog monitoring system, currently realized through the Fog Prober and its associated sensing agents, with standardized, open-source monitoring solutions.

VI. Experiments

In this section, we introduce an experiment conducted in our lab that aims to showcase the advantages of the discussed approach in a simple service placement scenario. The experiment starts with the definition of a microservice placement problem in a realistic Fog environment, proceeds with the solution of the problem performed by the *Problem Solver*, and ends with the test of the solution performed by both the *Simulator* and *Emulator*. The goal of the experiment is not to prove that the solution found performs better than others. We rather aim to show that the QoS promised by the solution suggested by the analytical model is attainable in a real production environment.

A. Fog scenario and placement problem

A Fog setting is characterized by i) a set of compute hosts connected via both local and wide area network links, ii) a pool of microservice-based services for which an optimal placement scheme must be devised that will guarantee the attainment of certain QoS levels, and iii) a number of users that generate requests towards the services.

To build a pool of services, we picked some heterogeneous and composable microservices that are typically employed in

[9]https://yaml.org/

TABLE III: Docker Images Size

Docker Image	Size (MB)
PRED	589
DIS	184
VREC_H	741
VREC_L	741
SENS	169
IREC	1270
DETS	169

IoT settings. Two types of microservices were considered: *high computing-demanding* and *low computing-demanding*. High computing-demanding microservices include the following:

- A **video recognition** microservice. It performs a video object detection and classification from a 1-second video input. This service outputs a numerical matrix that stores the position of the tracked object. It can be set to operate at two resolution levels: the *high-res (VREC_H)*, which produces a 1920×1080 matrix of points, and the *low-res (VREC_L)*, that outputs a 640×360 matrix.
- An **image recognition** microservice *(IREC)*. It exploits a pre-trained ML model[10] to perform object classification from an input image.
- A **system status prediction** microservice *(PRED)* that forecasts the status of a system and detects potential anomalies. The service makes use of an ML model pre-trained on historical time series.
- An **alarm** microservice *(DETS)*. It is an ML-powered service that, fed with an input value, raises an alarm based on a pre-set threshold.

The low computing-demanding category comprises:

- An **IoT sensor** microservice *(SENS)* that emulates a temperature sensor providing data at a given data rate. The emulated sensor outputs synthetic data collected from a public dataset[11].
- A **service discovery** microservice *(DIS)* which stores and handles a look-up table of microservices running in the Fog. The microservices look-up table leverage a Redis database [36] to store URLs of running microservices.

Docker implementations of the described microservices are publicly available in our repository[12]. Table III shows the size of all microservices Docker images.

Four *service chains* are built by composing microservices picked from the list above. A description of the workflow of each chain, along with the microservices that make up the chain, is given in Table IV.

For the purpose of the experiment, we considered the Fog federation topology depicted in Fig. 3. It includes two Fog data centers connected via Gigabit Ethernet links, which guarantee high stability and performance. Each data center comprises two hosts. Data center 1 (EDC1) features a switch that provides connection to the two hosts through 100 Mbps network links. The same configuration is reproduced in the Data center 2 (EDC2). All hosts participating in this experiment are equally powerful in terms of computing capacity. To make the network infrastructure more realistic, we derived some settings from the Italian GARR Network's statistics [37]. We assume that EDC1 is a data center located in the city of Bologna, while EDC2 is in Parma. According to statistics, the network link between EDC1 and EDC2 provides a 1 Gbps bandwidth and has an average RTT of about 10.9 ms. Furthermore, the links between fog nodes and their local switches provides a bandwidth as large as 100 Mbps, and an average delay of 2 ms and 4 ms for S1 and S2 respectively. The network configuration parameters are listed in Table V.

[10]https://github.com/DaveVoyles/docker-flask-image-recognition-sklearn

[11]https://github.com/limingwu8/Predictive-Maintenance/blob/master/dataset/csv/original/1711_TANK_TEMPERATURE.csv

[12]https://gitlab.com/unibo-microservices

TABLE IV: Service chains

Service chain ID	Microservices	Service workflow
VIDEO HI	DIS,VREC_H	DIS provides the URL of VREC_H which is eventually invoked to run the recognition
VIDEO LO	DIS,VREC_L	DIS provides the URL of VREC_L which is eventually invoked to run the recognition
IMAGE	DIS,IREC,PRED, DETS	DIS provides the URLs of IREC,PRED and DETS respectively. Those are invoked in a pipeline fashion
IOT	SENS,PRED, DETS	Microservices are invoked in a pipeline fashion

TABLE V: Network parameters

Network segment	Bandwidth	RTT
S1-S2	1 Gbps	10.9 ms
F1-F2	100 Mpbs	2 ms
F3-F4	100 Mpbs	4 ms

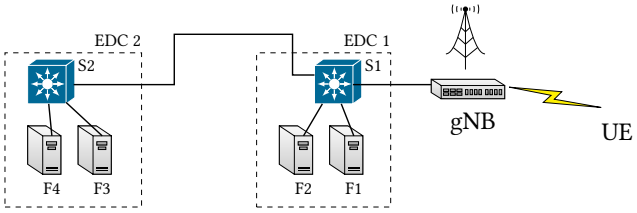


Fig. 3: Network topology used for the experiments

B. Experimental Results

In the fog scenario depicted, we intend to devise and test a solution to the problem of placing the service chains on the available hosts. First, the *Microservice Profiler* was invoked to benchmark the relevant microservices, with the objective of extracting meaningful performance indicators required by the *Problem Solver*. The hardware hosting the *Microservice Profiler* is the DELL R7525 datacenter that is equipped with a 96 x AMD EPYC 7413 24-Core Processor (2 Sockets). The metrics chosen for the assessment purpose are the mean and standard deviation of the service execution time. In particular, the **mean service execution time** is defined as the average time between when the user issues a request and when the service outputs a response. To automate this evaluation process, we configured the *Microservice Profiler* to stress each

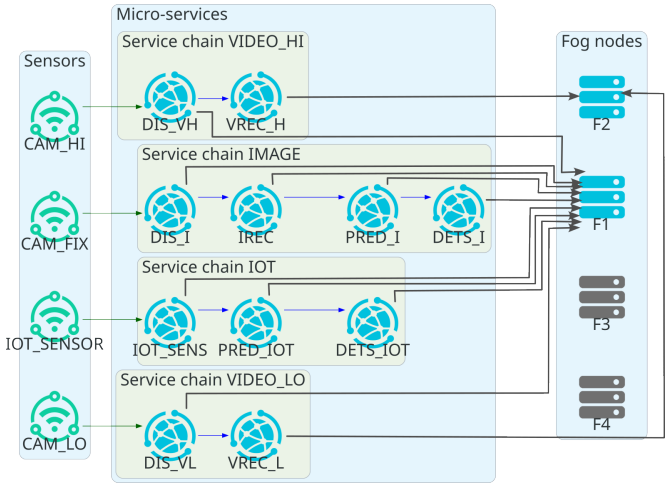


Fig. 4: Sample microservice placement scenario

TABLE VI: Characterization of microservices in terms of execution time

high computing-demanding services			
Service Name	**Service ID**	**avg. (s)**	**st.dev (s)**
Video recognition high-res	VREC_H	1.780	0.045
Video recognition low-res	VREC_L	0.751	0.027
Image recognition	IREC	0.182	0.167
Prediction service	PRED	0.053	0.006
Detection service	DETS	0.006	0.001
low computing-demanding services			
Service Name	**Service ID**	**avg. (s)**	**st.dev (s)**
IoT sensor	IOTS	0.003	0.001
Service discovery	DIS	0.009	0.001

microservice instance with 50 consecutive requests. Table VI shows the performance results obtained.

Secondly, we fed the *Problem solver* with all collected inputs. For our experiments, we use a population of 600 individuals and we let the population evolve for a maximum of 600 generations; the probabilities for the mutation operator is 0.4 and the crossover probability is 0.5; the selection operator operates with a 7-round tournament. All values have been derived from performance analyses of the GA algorithm available in the literature [21]. The problem solver outputs the microservice placement scheme depicted in Fig. 4 Finally, the placement scheme was handed over to both the *Simulator* and the *Emulator*. For the purpose of the experiment, both components were hosted and ran on the previously mentioned hardware, that is, the DELL R7525 datacenter.

In Fig. 5a, we present the average service response time predicted by the *Problem Solver* using two different heuristics (Genetic Algorithm heuristics and AMPL-based solver using K-Nitro) with the observed response time in the *Simulator*. We also split the response time into waiting time, service time, and network delay. We observe that the output provided by the theoretical model and the simulation are quite similar, with an error that keeps below 2%. Furthermore, we observe that both implementation solvers achieve the same solution for the placement problem, predicting identical performance.

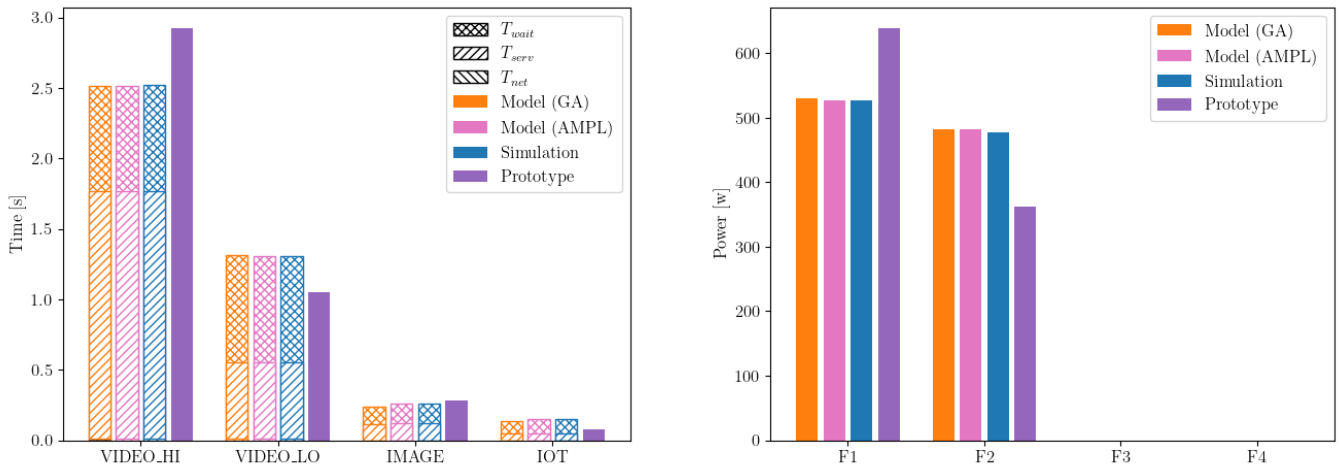
In the same figure, we also depicted the service response times observed in the *Emulator*. The comparison demonstrates that the simulation effectively captures the main characteristics of the services, despite an error range of 10% to 20%. The largest discrepancy occurs in the VIDEO HI service chain, likely due to the interaction between heavy load services co-located on the same nodes. It is worth to note, however, that the error between the simulator and the emulator is, even in the worst case, in the order of 70% with respect to the standard deviation of the service response time. However, part of the error is due to the containerization aspects that the simulation fails to capture. Specifically, there are Docker dynamics at runtime that are not yet considered by both the model and the simulation, as they require further study.

In contrast, Fig. 5b presents a comparison of the fog node power consumption based on the simulation, model, and prototype. Prototype data was collected using the *vmstat* command on a single Fog node while the workload was running.

In Fig. 6, we report the service response time distribution collected during the emulation. We observe that the VIDEO HI chain's response time is characterized by a high variability, as also witnessed by the many outliers (represented by circles in the chart). The VIDEO LO chain shows a similar behavior, even though the variability is less marked. The other two service chains (IMAGE and IOT) are characterized by a more stable response time.

Finally, in the Table VII we report the statistics of the execution times of all the platform components involved in the experiment. The statistics were collected after 50 runs for which the same placement problem is presented to the platform. The *Simulator* accomplishes its task in a very short time. The *Problem Solver* reaches a solution very quickly in the case where the Genetic Algorithm heuristic is used. On the other hand, the time pro the problem solver based on AMPL are significantly longer, demonstrating the clear performance benefit of the heuristic over the commercial solver. A scalability analysis conducted on the algorithm that powers the *Problem Solver* [38] proves that the solution can be attained in a limited lapse of time also in the case of more complex problems, while preliminary tests on the solver suggests that, for large problem instances, the AMPL-based solver takes too much time to converge (the solver is unable to reach an optimal solution within the configured time limit of 60 minutes). Regarding the Emulator, most of the execution time (70%) is spent by the Containernet to prepare the emulation, i.e., to set up the network, instantiate the nodes and deploy the Docker containers. The rest is for actual emulation of end user requests and services response.

The results obtained confirm that the designed system can fulfill needs of optimal microservice placement in a fog scenario. In particular, the system can cope with the dynamics of a context where the boundary conditions may suddenly change (e.g., new services need to be placed, services have to face a peak of requests, nodes may experience HW/SW failures, etc.): in those cases, a service re-placement can be promptly run that will provide a robust solution in a short time.



(a) Response time

(b) Fog node power consumption

Fig. 5: Comparison of model, simulation and prototype

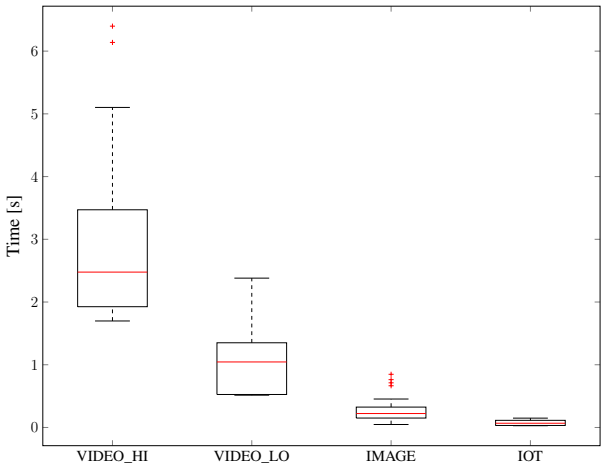


Fig. 6: Microservices chain response time distribution observed the emulation

TABLE VII: Platform components execution time

	Average execution time (s)	**std (s)**
Problem Solver (GA)	6.81	0.42
Problem Solver (AMPL)	79.5	0.87
Simulator	0.89	0.06
Emulator	11.33	0.09

VII. Concluding Remarks

In the literature, placement strategies in Fog environments are typically validated either through simulation or through small-scale testbeds, each offering distinct advantages and limitations. Building on this gap, this work introduced *OptiFog*, a framework and accompanying software platform that combines the strengths of both approaches. OptiFog supports developers in designing and validating microservice placement algorithms before deploying them in real Fog infrastructures, and provides guarantees on the expected QoS even when scaling to medium-to-large deployments.

We presented the OptiFog architecture, detailing its main components, and conducted an experiment based on a realistic IoT microservice placement scenario, including both sub-optimal and optimal solutions. The results showed that OptiFog significantly simplifies the development and evaluation of placement strategies, while offering insights into the consistency and reliability of the computed solutions. Notably, thanks to its emulator, OptiFog enabled us to identify a small discrepancy between the predicted QoS and the QoS observed in the testbed, a capability that, to our knowledge, no existing placement framework currently provides. Overall, the experiment demonstrated that OptiFog can effectively handle the unpredictable dynamics that characterize fog deployments. In particular, our work first represents a proof of concept, and the current OptiFog prototype has been tested only in small-scale settings with a limited number of nodes and services. Second, although all individual components rely on robust technologies (e.g., OMNeT++, Docker), they have been evaluated separately; comprehensive stress tests involving their integrated orchestration are still needed.

Looking ahead, we plan to pursue two main research directions. On the one hand, we will extend the prototype to support larger and more complex deployments and evaluate OptiFog in real (not only realistic) fog environments. In particular, future experiments will assess its effectiveness under dynamic network conditions, beyond the static parameters of the Italian GARR network used in the current study. On the other hand, we aim to enhance the orchestrator to automate the deployment of microservices across large-scale fog–cloud infrastructures and to better coordinate the execution of the platform's components.

Acknowledgments

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART") CUP: J33C22002880001

REFERENCES

[1] R. Mahmud, R. Kotagiri, and R. Buyya, *Fog Computing: A Taxonomy, Survey and Future Directions*. Singapore: Springer Singapore, 2018, pp. 103–130. [Online]. Available: https://doi.org/10.1007/978-981-10-5861-5_5

[2] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan, "Fog computing: Survey of trends, architectures, requirements, and research directions," *IEEE Access*, vol. 6, pp. 47 980–48 009, 2018.

[3] F. A. Salaht, F. Desprez, and A. Lebre, "An overview of service placement problem in fog and edge computing," *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3391196

[4] F. Sarkohaki and M. Sharifi, "Service placement in fog–cloud computing environments: a comprehensive literature review," *J. Supercomput.*, vol. 80, no. 12, p. 17790–17822, May 2024. [Online]. Available: https://doi.org/10.1007/s11227-024-06151-4

[5] M. M. Islam, F. Ramezani, H. Y. Lu, and M. Naderpour, "Optimal placement of applications in the fog environment: A systematic literature review," *Journal of Parallel and Distributed Computing*, vol. 174, pp. 46–69, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731522002465

[6] S. Pallewatta, V. Kostakos, and R. Buyya, "QoS-aware placement of microservices-based IoT applications in Fog computing environments," *Future Generation Computer Systems*, vol. 131, pp. 121–136, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X22000206

[7] M. R. Hossain, M. Whaiduzzaman, A. Barros, and C. Fidge, "Dynamic microservice placement in multi-tier fog networks," *Internet of Things*, vol. 26, p. 101224, 2024.

[8] A. Yousefpour, A. Patil, G. Ishigaki, I. Kim, X. Wang, H. C. Cankaya, Q. Zhang, W. Xie, and J. P. Jue, "Fogplan: A lightweight qos-aware dynamic fog service provisioning framework," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 5080–5096, 2019.

[9] Q. Deng, M. Goudarzi, and R. Buyya, "Fogbus2: a lightweight and distributed container-based framework for integration of iot-enabled systems with edge and cloud computing," in *Proc. of Workshop on Big Data in Emergent Distributed Environments*, ser. BiDEDE. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3460866.3461768

[10] S. Pallewatta, V. Kostakos, and R. Buyya, "MicroFog: A Framework for Scalable Placement of Microservices-based IoT Applications in Federated Fog Environments," 2023.

[11] A. Marchese and O. Tomarchio, "Enhancing the kubernetes platform with a load-aware orchestration strategy," *SN Comput. Sci.*, vol. 6, no. 3, Feb. 2025. [Online]. Available: https://doi.org/10.1007/s42979-025-03712-z

[12] S. B. Nath, S. Chattopadhyay, R. Karmakar, S. K. Addya, S. Chakraborty, and S. K. Ghosh, "Containerized deployment of micro-services in fog devices: a reinforcement learning-based approach," *J. Supercomput.*, vol. 78, no. 5, p. 6817–6845, Apr. 2022. [Online]. Available: https://doi.org/10.1007/s11227-021-04135-2

[13] R. Mahmud and A. N. Toosi, "Con-Pi: A Distributed Container-Based Edge and Fog Computing Framework," *IEEE Internet of Things Journal*, vol. 9, no. 6, pp. 4125–4138, 2022.

[14] B. Alturki, S. Reiff-Marganiec, C. Perera, and S. De, "Exploring the Effectiveness of Service Decomposition in Fog Computing Architecture for the Internet of Things," *IEEE Transactions on Sustainable Computing*, vol. 7, no. 2, pp. 299–312, 2022.

[15] S. Deng, Z. Xiang, J. Taheri, M. A. Khoshkholghi, J. Yin, A. Y. Zomaya, and S. Dustdar, "Optimal application deployment in resource constrained distributed edges," *IEEE Transactions on Mobile Computing*, vol. 20, no. 5, pp. 1907–1923, 2021.

[16] T. Huang, W. Lin, C. Xiong, R. Pan, and J. Huang, "An Ant Colony Optimization-Based Multiobjective Service Replicas Placement Strategy for Fog Computing," *IEEE Transactions on Cybernetics*, vol. 51, no. 11, pp. 5595–5608, 2021.

[17] M. Ghobaei-Arani and A. Shahidinejad, "A cost-efficient iot service placement approach using whale optimization algorithm in fog computing environment," *Expert Systems with Applications*, vol. 200, p. 117012, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417422004304

[18] C. Guerrero, I. Lera, and C. Juiz, "Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures," *Future Gener. Comput. Syst.*, vol. 97, no. C, p. 131–144, Aug 2019.

[19] Z. Ding, S. Wang, and C. Jiang, "Kubernetes-oriented microservice placement with dynamic resource allocation," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1777–1793, 2023.

[20] E. Bozkaya, M. Erel-Özçevik, T. Bilen, and Y. Özçevik, "Proof of evaluation-based energy and delay aware computation offloading for digital twin edge network," *Ad Hoc Networks*, vol. 149, p. 103254, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870523001749

[21] C. Canali, R. Lancellotti, and R. Mescoli, "An analysis of genetic algorithms to support the management of edge computing infrastructures," in *2024 22nd International Symposium on Network Computing and Applications (NCA)*, 2024, pp. 140–147.

[22] M. Goudarzi, M. S. Palaniswami, and R. Buyya, "A distributed deep reinforcement learning technique for application placement in edge and fog computing environments," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021.

[23] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "Demand-driven deep reinforcement learning for scalable fog and service placement," *IEEE Transactions on Services Computing*, vol. 15, no. 5, pp. 2671–2684, 2022.

[24] I. Sarkar, M. Adhikari, S. Kumar, and V. G. Menon, "Deep reinforcement learning for intelligent service provisioning in software-defined industrial fog networks," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 16 953–16 961, 2022.

[25] C. Canali, G. Di Modica, R. Lancellotti, and D. Scotece, "Optimal Placement of Micro-services Chains in a Fog Infrastructure," in *Proc. of International Conference on Cloud Computing and Services Science (CLOSER)*, Apr. 2022.

[26] D. Ardagna, M. Ciavotta, R. Lancellotti, and M. Guerriero, "A hierarchical receding horizon algorithm for qos-driven control of multi-iaas applications," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2018.

[27] X. Xia, H. Qiu, X. Xu, and Y. Zhang, "Multi-objective workflow scheduling based on genetic algorithm in cloud environment," *Information Sciences*, vol. 606, pp. 38–59, 2022.

[28] I. M. Ali, K. M. Sallam, N. Moustafa, R. Chakraborty, M. Ryan, and K.-K. R. Choo, "An automated task scheduling model using non-dominated sorting genetic algorithm ii for fog-cloud systems," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2294–2308, 2022.

[29] C. Canali, C. Gazzotti, R. Lancellotti, and F. Schena, "Placement of IoT Microservices in Fog Computing Systems: A Comparison of Heuristics," *Algorithms*, vol. 16, no. 9, 2023.

[30] Apache, "Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html, 2023, [Online; accessed 01-November-2023].

[31] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017.

[32] O. Tomarchio, D. Calcaterra, G. Di Modica, and P. Mazzaglia, "Torch: a tosca-based orchestrator of multi-cloud containerised applications," *Journal of Grid Computing*, vol. 19, no. 1, 2021.

[33] OASIS, "Topology and Orchestration Specification for Cloud Applications Version 1.3," https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html, Nov. 2020, Last accessed: 25-07-2024.

[34] OMG, "Business Process Model and Notation (BPMN 2.0)," http://www.omg.org/spec/BPMN/2.0/, Jan. 2011, Last accessed on 25-07-2024.

[35] M. Peuster, H. Karl, and S. van Rossem, "Medicine: Rapid prototyping of production-ready network services in multi-pop environments," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2016, pp. 148–153.

[36] Redis, "Redis Database," https://redis.io/, 2023, [Online; accessed 01-January-2023].

[37] GARR, "GINS Home Statistics," https://gins.garr.it/home_statistics.php, 2023, [Online; accessed 01-January-2023].

[38] C. Canali, G. Di Modica, R. Lancellotti, S. Rossi, and D. Scotece, "A validated performance model for micro-services placement in fog systems," *SN Comput. Sci.*, vol. 4, no. 4, may 2023.

Claudia Canali is an associate professor at the Department of Engineering "Enzo Ferrari" in the University of Modena and Reggio Emilia since 2018. She received the Laurea Degree, summa cum laude, in computer engineering from the same University in 2002 and the Ph.D. in Information Technology from the University of Parma in 2006. Her research interests include edge and distributed systems, fog and cloud computing, and policies addressing the gender gap in ICT and STEM.

Giuseppe Di Modica received the Ph.D. in Computer Science and Telecommunication Engineering from the University of Catania, Italy. He is an associate professor with the Department of Computer Science Engineering at the University of Bologna, Italy. His research interests include Cloud Computing and multi Cloud, Edge/Fog computing, Big Data, Virtualization technologies, Internet of Things, and engineering of distributed systems in verticals such as Industry and Smart Cities.

Francesco Faenza is a research fellow at the University of Modena and Reggio Emilia since March 2024. He holds a Ph.D. from the E4E School of the University of Modena and Reggio Emilia in 2024, and also earned a Master's degree in Computer Engineering from the same institution in 2019. His research interests focus on edge and distributed systems, and DevOps methodologies. He is actively involved in several Erasmus+ projects. He has 10 years of experience in the private sector as a system administrator. He is a member of IEEE and ACM.

Luca Foschini (Senior Member) received a Ph.D. degree in computer science engineering from the University of Bologna, Italy, where he is now a full professor of distributed systems. 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.

Riccardo Lancellotti is an associate professor at the Department of Engineering "Enzo Ferrari" in the University of Modena and Reggio Emilia since 2015. He received the Laurea Degree in computer engineering from the University of Modena and Reggio Emilia in 2001 and the Ph.D. in computer engineering from the University of Roma "Tor Vergata" in 2003. His research interests include geographically distributed systems, fog and cloud computing. On these topics he published more than 100 papers in international journals and conferences. He is a member of IEEE and of ACM.

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. 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.