Emulating the SDN-Enabled Computing Continuum through Lightweight Virtualization

José Gómez-delaHiz* *Student Member, IEEE*, Juan Luis Herrera†, Sergio Laso*, Javier Berrocal* *Member, IEEE*, Jaime Galán-Jiménez* *Member, IEEE*

***Abstract*—In recent years, the number of Internet-connected devices has increased notably, leading to a significant rise in data traffic. This increase has been fostered by the Internet of Things paradigm, the use of microservices architectures in application development, and the ability to deploy these applications across various layers in the Computing Continuum (including Fog, Edge, and Cloud layers). Consequently, choosing the right deployment strategy has become essential for network operators and developers, especially in intensive domains such as smart cities. In this work, we introduce an emulation framework that allows developers and operators to decide how to deploy networks, computing devices, and applications in a Computing Continuum environment, ensuring compliance with established Quality of Service standards. This framework supports both IP and SDN network paradigms and is highly adaptable to different scenarios due to its use of container-based virtualization. Furthermore, the SDN paradigm provides flexibility, enabling the implementation of a service discovery feature that simplifies communication between end devices and services. Evaluations conducted in a realistic smart city scenario show that this framework can be extended and applied to a wide range of situations and configurations, meeting the needs of the research community in the Computing Continuum domain.**

***Index Terms*—Computing Continuum, SDN, Fog, IoT, Service Discovery**

I. INTRODUCTION

The Internet of Things (IoT) is a rapidly evolving technology with substantial technical, social, and economic importance. Nowadays, IoT devices are frequently used in diverse environments to carry out daily tasks. In 2023, the number of devices connected to IP networks is approximately 29.3 billion [1]. The emergence of new applications that demand strict Quality of Service (QoS) levels, especially in smart cities, poses a challenge for network operators in managing large volumes of data. In an IoT-based network scenario, optimizing the QoS involves considering three layers: the network layer, the computing layer, and the applications and services layer.

The application and services layer pertains to IoT applications designed using the Microservices Architecture (MSA) pattern. MSA applications are characterized by their division into small, loosely-coupled microservices [2]. When simple tasks work together, they can manage more complex functionalities. The sequence of ordered microservices that make up each application functionality is referred to as a workflow, with each user request linked to a specific workflow [3]. MSAs can be deployed across distributed infrastructure, providing higher flexibility [2]. The problem of optimally placing a set of microservices within a Computing Continuum infrastructure, paramount to optimize application QoS, is referred to in the literature as the Decentralized Computation Distribution Problem (DCDP) [3]. The Computing Continuum is a distributed computing paradigm that extends the cloud with user-proximate computing resources in the fog and edge layers, providing heterogeneous computing capabilities across the network to reduce latency, improve resilience, and decrease energy consumption in distributed applications [4].

The computing layer emphasizes the efficient distribution of services. Traditionally, cloud computing was used to deploy IoT applications. However, since cloud servers are often distant from IoT devices, it is beneficial to shift microservices to the Computing Continuum, particularly to nodes nearer to users, to decrease application response time [3]. Consequently, the placement of Computing Continuum nodes becomes vital for optimizing the QoS. The challenge of optimally positioning these nodes was initially addressed in the context of the fog layer and is referred to as the Fog Node Placement Problem (FNPP) [5].

Finally, the network layer manages communications. The adoption of technologies like Software-Defined Networks (SDN) and Network Function Virtualization (NFV) aligns well with the concept of decentralizing IoT applications through the MSA paradigm [6]. Therefore, the Controller Placement Problem (CPP) [7] is a critical challenge in this context, focusing on positioning the SDN controller to ensure optimal QoS for the switches.

Developers of intensive IoT applications are interested in reducing the deployment cost and improving the network performance to meet their QoS targets. To achieve such goals, different authors have proposed methods to solve the CPP, DCDP, and FNPP [3], [5]–[7]. However, to properly evaluate how these methods affect each of the three layers, developers need to execute their software over realistic scenarios. To the best of our knowledge, existing solutions are unable to

* Dept. of Computer Systems and Telematics Engineering, University of Extremadura, Spain
† Distributed Systems Group, Technische Universität Wien, Vienna, Austria
(E-mail: jagomezdh@unex.es, j.gonzalez@dsg.tuwien.ac.at, slasom@unex.es, jberolm@unex.es, jaime@unex.es)
Corresponding author: e-mail: jagomezdh@unex.es

DOI: 10.36244/ICJ.2026.2.2

consider all three layers, are simulated testbeds that cannot be integrated with real application and network software, are tied to a specific method or solver, or need to be configured from scratch in each execution [8]–[10].

To address this need, this work presents an emulation framework, named Emulation Framework for the Computing Continuum (EFCC), to automate the configuration and generation of emulated Computing Continuum testbeds in all three layers. Researchers can use EFCC to evaluate their solutions, ad-hoc or using different methods, for the CPP, DCDP, FNPP or any combination of them. Moreover, as the implementation of service discovery is considered a key feature for applications in the SDN-enabled Computing Continuum [3], EFCC provides built-in support for transparent, network-level service discovery. The main contributions of this work are:

- The proposal of EFCC, a framework for emulating scenarios involving the deployment of MSA-based IoT applications within SDN-based Computing Continuum environments.
- The implementation of transparent service discovery in the SDN controller. EFCC can automatically perform service discovery at the network level through workflow information processing without affecting traditional network traffic, and with minimal effects on QoS.
- Proposal of the DADOSIM scenario modeling format, and the generation of a DADOSIM dataset with scenarios involving different sizes, devices, microservices, and workflows.
- The evaluation of EFCC, including the emulation capabilities, the transparent service discovery functionality, over diverse realistic smart cities-based IoT scenarios, and an unified emulation across cloud-edge-IoT.
- A scalability analysis of emulation performance in Kathará, including the deployment time and the memory usage for each of the tests.

The remainder of this paper is structured as follows. Section II introduces the existing solutions and their limitations. Section III describes the proposed framework and the modules it is composed of. Section IV analyzes the obtained results over realistic use cases. Finally, Section V concludes the work.

II. Related Work

In the Computing Continuum environment, several solutions offer valuable contributions at different scales. These include extensive simulators and emulators for the cloud, fog, and edge layers, which aid in decision-making and provide critical control information about application performance and network infrastructure management. This section reviews these simulators and emulators for the Computing Continuum.

The first work analyzed is FogSim [8], a Java-based simulator for the Fog layer. FogSim evaluates two EDF-based resource management strategies: Distributed EDF for Fog (DEF) and Profit-aware Distributed EDF for Fog (PDEF). DEF improves resource allocation using linear programming while considering latency, processing cost, storage capacity, and energy consumption, whereas PDEF balances dynamic pricing and demand to optimize cost efficiency. Although FogSim enables the analysis of Fog computing scenarios under these strategies, it is tightly coupled to them, requiring explicit implementation to evaluate alternative approaches.

Next, we analyze CloudSim [9], a simulation framework for cloud computing environments. CloudSim allows the configuration of diverse scenarios with different resource settings, allocation policies, and workloads, and supports the evaluation of metrics such as response time, throughput, power consumption, and resource utilization. While this flexibility enables detailed performance analysis and optimization of cloud resource management, CloudSim is limited to traditional cloud computing, lacking support for the Computing Continuum and distributed layers. Moreover, it only supports conventional IP networks, preventing the simulation of SDN-based scenarios, and, as a simulator, shares the same application execution limitations as FogSim.

Another Computing Continuum simulation proposal is YAFS (Yet Another Fog Simulator) [10], a Python-based simulator for Fog and Edge environments. YAFS supports the modelling of Fog infrastructures, including Edge devices, computing resources, and network elements, and allows the evaluation of applications, workloads, and resource management policies through a flexible API for defining topologies and resource configurations. It enables the analysis of metrics such as latency, response time, resource utilization, and power consumption. However, as a simulator, YAFS shares the inherent limitations in application execution and lacks support for the SDN paradigm.

Our proposed EFCC framework offers substantial differences compared to previously analyzed tools. EFCC enables the emulation of complete systems, including the applications to be tested, network elements (such as switches and SDN controllers), and terminal devices (hosts). Utilizing Docker containerization, all these components run real software, yielding results that are more reflective of actual conditions than those obtained with simulation tools. Applications can be divided into microservices following an MSA infrastructure, with some microservices capable of communicating with IoT devices through a Service Function Chaining (SFC) [11] workflow architecture. Additionally, EFCC focuses on the SDN network paradigm and can implement transparent service discovery within the SDN controller, allowing devices to communicate with microservices without needing to know their exact locations beforehand without significantly impacting performance metrics.

III. EFCC Framework

In this section, we detail the proposed EFCC framework by outlining its different modules. The scenario construction begins with a data input module, followed by a parser that builds the infrastructure. This setup allows for testing both network and application performance. EFCC consists of four main modules: (i) the DADOSIM scenario format (detailed in Section III-A), (ii) the parser algorithm (explained in Section III-B), (iii) the Kathará lab with its associated Docker containers (described in Section III-C), and (iv) the POX controller in an SDN network scenario, including service

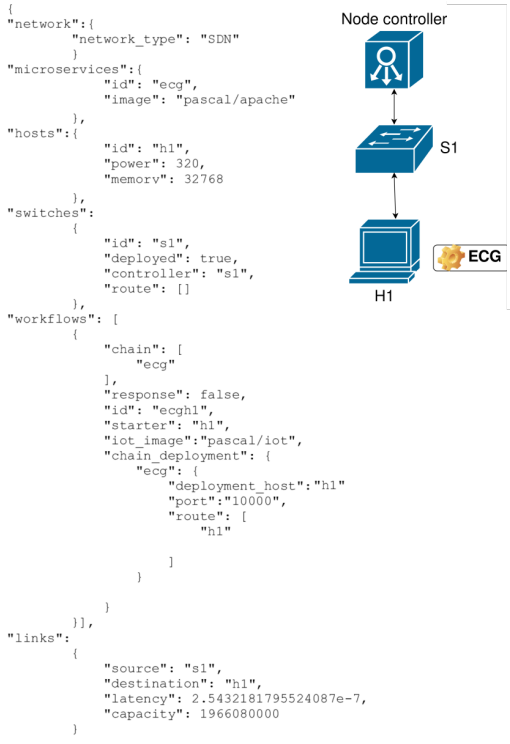


Fig. 1. Scenario example, including its description in the DADOSIM format.

discovery functionality and behavior in traditional IP networks (detailed in Sections III-D and III-E, respectively).

A. DADOSIM scenario format

Within EFCC, a *scenario file* expresses the complete configuration of an emulated testbed, including the deployment and configuration of the application, computing, and network layers. Scenario files have a JSON format that follows a specific schema named DADOSIM, as it is similar to the input format used by the DADO optimization framework [6]. An example EFCC scenario file, along with a visual representation of the scenario, is provided in Fig. 1. The DADOSIM format is structured in 6 blocks, which are described in the following:

- **Network**: This block defines the paradigm used by the network layer of the scenario. EFCC supports both traditional IP networks and SDN ones.
- **Microservices**: This block contains the list of the different microservices defined in the application's MSA. Each microservice must have its own unique identifier and define the Docker image that implements the microservice. This can refer to images in the local Docker image registry, or remote images that will be pulled when the emulation is started.
- **Hosts**: This block defines the computing layer of the scenario, i.e., its computing devices or hosts. Each must have a unique identifier, and its specifications in terms of power (in CPU percentage) and RAM (in bits).
- **Switches**: This block defines the SDN switches or IP switches and routers that integrate the networking fabric. A unique identifier also defines each switch. Moreover, the *deployed* attribute defines whether an SDN controller is or is not deployed to the switch, while the *controller* attribute defines the switch co-located with the SDN controller that the switch will communicate with. If the switch needs to communicate with another controller, the route used for its traffic can be defined in the *route* attribute. If no route is defined, the shortest path routing algorithm will be used, leveraging latency length metric.
- **Workflows**: This block defines the different workflows requested in the scenario. Each workflow must have a unique identifier. Moreover, they contain information about the IoT device requesting the workflow (*starter*), all the microservices they request (*chain*), and whether the data provided as output by the final microservice should be routed to the requesting IoT device (*response*). The workflow also defines the Docker image used by the requesting IoT device (*iot_image*), and details the deployment of the microservices (*chain_deployment*). For each requested microservice, the workflow defines the ID of the host in which it is deployed (*deployment_host*) and its port. Moreover, the path used to access this microservice from the previous one can be specified.
- **Links**: This final block defines the links that will connect the hosts and switches. Links do not need to have a unique identifier and are bidirectional, and their latency (in seconds) and capacity (in bytes) must be specified.

B. EFCC parser

EFCC automatically generates the container definitions, the underlying network links, and their configuration to automate the creation of a testbed that accurately represents a Computing Continuum scenario with the desired MSA-based IoT application running over an IP or SDN network. The remainder of this subsection details how the parser creates and configures containers, workflows, and network links.

1) Container creation: EFCC generates containers for some elements described in the DADOSIM files: switches, hosts, and microservices, although they are handled differently depending on their type. Beginning with switches, for each of the defined switches, a container is generated, which executes Quagga if the network is IP-based, and OpenVSwitch if the network is SDN-based. For example, switches *S1*, *S2*, *S3*, and *S4* from Fig. 2 become a separate containers in Fig. 3. EFCC automatically generates the necessary configurations in both cases (e.g. routing protocol daemons, bridge generation, execution mode) so the developer or operator does not need to manually do so. Moreover, for SDN networks, the SDN controller is also provided as a container, executing the custom POX EFCC controller described in Section III-D. Among the available SDN controllers, POX was selected over more recent alternatives due to its ease of understanding, reimplementation, and modification, to facilitate the reuse of the proposed code.

Hosts and microservices are handled differently: each microservice in MSA-based applications is packaged as a separate Docker image [2], and should be instantiated as a separate container. Hence, a host running multiple microservices cannot

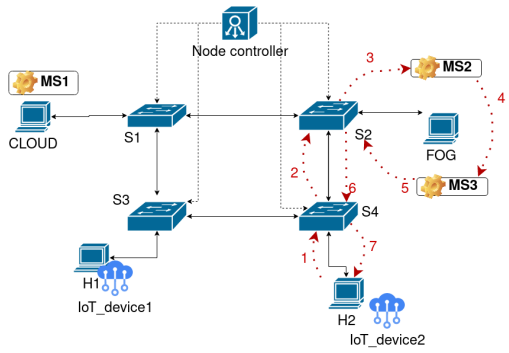


Fig. 2. Example scenario with 4 SDN switches, an SDN controller, 4 hosts, 3 microservices, 2 IoT devices, and 1 workflow.

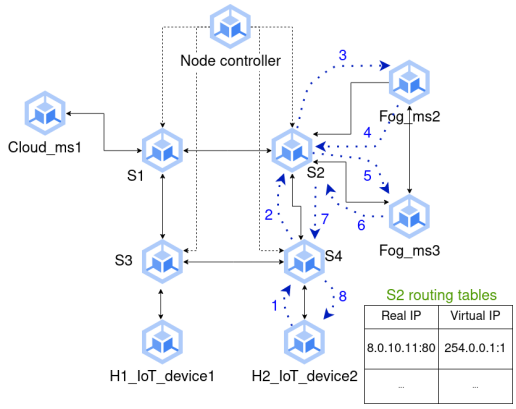


Fig. 3. Containers generated by EFCC from the scenario depicted in Fig. 2.

be directly converted to a single container. Instead, a container is generated for every microservice executed by a host. For example, a host running a single microservice, like the *Cloud* node in Fig. 2, becomes a single container (*Cloud_ms1*) in Fig. 3, while a host running two microservices (*Fog* in Fig. 2) becomes two containers (*Fog_ms2* and *Fog_ms3* in Fig. 3). Nonetheless, the hardware considerations (i.e. CPU, RAM) are still applied.

The networking across the generated containers is adjusted to ensure different containers that represent different microservices executed by the same host can communicate as if they were executed in the same machine. Furthermore, the general configurations for these containers (e.g. IP addresses, routing gateways, etc.) are automatically configured as well. Devices that request workflows, such as IoT devices, have another container created for them, using the image specified as the IoT image for the workflow in the scenario file, and are also automatically configured. This is shown as hosts *H1* and *H2* from Fig. 2 becoming a container each in Fig. 3.

2) Workflows: Workflows are the model with which IoT devices request application functionalities involving one or more microservices in order. The goal is to fulfil the request initiated by the IoT device, designated as the source node. In Fig. 2, a representation of a workflow is shown in red color. This workflow is requested by the IoT device of host *H2* (*H2_IoT_device2*) and requires microservices *MS2* and *MS3*

Algorithm 1 Pseudocode of the link creation subroutine

```
1: for all links ∈ scenario do
2:     if link.source is switch and link.destination is switch then
3:         domain ← createDomain(link.endA.ID + link.endB.ID)
4:         link.endA.registerDomain(domain)
5:         link.endB.registerDomain(domain)
6:     else
7:         host ← link.getHost()
8:         if not domain then
9:             domain ← createDomain(link.getS().ID + host.ID)
10:        end if
11:        link.getS().registerDomain(domain)
12:        if (host.countIoT() + host.countMicroservices()) > 1 then
13:            if not innerDomain then
14:                innerDomain ← createDomain(host.ID)
15:            end if
16:            for all containers ∈ host do
17:                container.registerDomain(innerDomain)
18:                container.registerDomain(domain)
19:                if not container.defaultGw then
20:                    container.defaultGw ← link.getS().IP(domain)
21:                end if
22:            end for
23:        else
24:            host.container.registerDomain(domain)
25:            if not host.container.defaultGw then
26:                host.container.defaultGw ← link.getS().IP(domain)
27:            end if
28:        end if
29:    end if
30: end for
```

located in the *Fog* node.

In the scenario file, users can specify the port where each microservice will be deployed on the host and the path the request will take to reach the host. This routing information is stored in a separate JSON file, which serves as the input for configuring the custom SDN controller. If a microservice lacks a predefined route, the shortest path between the source and destination hosts is determined.

3) Networking links and domains: The next part of the parser creates the links across the containers created in the previous parts. As some hosts are represented by multiple containers, EFCC generates not only the links stated in the scenario file, but also *virtual links* to connect different containers that represent the same host. Moreover, some individual links defined in the scenario file can become multiple links: the link between *S2* and *Fog* in Fig. 2 becomes two links, *S2-Fog_ms2* and *S2-Fog_ms3*, in Fig. 3. As EFCC leverages the Kathará [12] network emulator to create the testbed, links are represented by Kathará *domains*. A domain can be seen as the representation of a network cable: all network interfaces connected to the same domain are considered to be directly connected to each other.

As link creation is a complex process, the pseudocode for the link creation subroutine in EFCC is described by Alg. 1. This subroutine allows achieving network connectivity by creating and associating different domains with the containers that will act as switches, IoT devices, and microservices. First, if the link connects two switches, a domain is created and associated with both (lines 2-5). On the other hand, if the link connects a switch to a host (lines 6-23), different considerations are taken. First, a domain for the link is created using the switch and host's IDs (lines 8-10), and associated with the host (line 11). Then, it is necessary to know if the host has multiple containers to adapt the link generation (lines 12-23). If so, a virtual link is created through a virtual domain

(lines 13-15), and all containers are associated with the host are connected to both the virtual domain and the link's domain (lines 16-18). Moreover, if the containers had no gateway, the switch is defined as their gateway (lines 19-21). On the other hand, if the host has only one container, it is just connected to the link domain (line 24), and the switch serves as its gateway if no prior one existed (lines 25-27). This ensures all containers belonging to the same host are connected to the same virtual domain, separate from the rest of the network, as well as link multiplexing for switches connected to hosts that become multiple containers.

Each domain is configured as an independent IP subnet with automatic addressing. When a host joins a domain, EFCC creates the domain interface and assigns an IP address, allowing containers to be multi-homed (one interface per connected domain) and supporting scalable deployments. For example, in Fig. 3, *S2* is connected to 5 domains (*S2*-controller, *S2-S1*, *S2-S4*, *S2-Fog_ms2*, and *S2-Fog_ms3*), and thus has 5 different IP addresses, each belonging to a different domain.

C. Kathará laboratory and Docker containers

Kathará is a network emulation tool that allows to create and configure an emulated network topology that interconnects containers leveraging Docker technology [12], widely used for realistic network simulations [13], [14]. EFCC automates the process of building an emulated network testbed on top of Kathará by automatically configuring the network domains, interfaces, and containers. To do so, the EFCC parser generates a Kathará laboratory: a configuration of Kathará that allows replicable executions of a network environment, including the commands to be executed in each container, the network connectivity, the computing constraints of each container, and other internal configurations.

To create and configure containers for the topology, the parser generates a configuration file for each host named *hostname.startup*. This file is used by Kathará, enabling EFCC to specify the actions to be executed when each container starts. The configuration varies depending on the type of node being considered (IoT devices, microservices, or switches).

- **IoT devices and microservices**: IoT devices and microservices containers will configure their network interfaces using the `ifconfig` command, specifying the IP address, netmask, and broadcast addresses for each interface. The Traffic Control Linux tool (`tc` command) will be employed to limit the bandwidth and delay of the respective interfaces. Lastly, the `route` command will be used to set the default gateway for the host.
- **Switches**: The configuration file for a switch is more complex, as it includes a series of OpenVSwitch configuration commands. These commands connect the switch's interfaces with the necessary virtual ports. Network interfaces will be assigned their IP addresses using the `ifconfig` command, while their delay and bandwidth will be managed using the Traffic Control tool (`tc`).

EFCC then generates a Kathará laboratory configuration (*lab.conf*) file, which defines the configuration of the network topology to be created and emulated. After starting the scenario in Kathará, a Python script will be executed to verify that all containers have been correctly deployed. If any errors are detected, the script will restart the environment.

D. POX controller

The EFCC framework can deploy SDN-based testbeds, and therefore, a controller for the SDN network is required. EFCC utilizes a customized version of the POX controller to serve as the SDN controller due to its flexibility, extendability, and adaptability for different use cases. Additionally, this customized SDN controller implements a service discovery mechanism using workflows, enabling IoT devices to communicate with microservices without needing to know their specific locations.

1) Basic controller behavior: The following outlines the network functions required to ensure connectivity between IoT devices and microservices. The controller is responsible for installing the necessary flow rules in the SDN switches, enabling requests from IoT devices to reach microservices and allowing microservices to communicate with each other.

Communication between switches and the controller is facilitated by the OpenFlow protocol, which allows traffic flows to be managed within the SDN network. Routing decisions can be customized by the framework user, as described in Section III-A. If no routing is specified, the shortest path algorithm is applied instead. The EFCC parser will write all routing information for each path (IPs and ports) to an external file named *data.json*, which the POX controller can interpret.

2) Service discovery and virtual traffic: EFCC distinguishes between virtual traffic and real traffic. Real traffic includes traditional network traffic directed towards a known destination. Virtual traffic, on the other hand, requires service discovery since the destination address (including IP address and port) is not known in advance. This type of traffic typically involves requests for services from IoT devices, where the destination is a specific microservice within a particular workflow. Virtual traffic must use the TCP protocol , and target a virtual IP address and destination port. The POX controller then automatically handles service discovery, routing the traffic to the container hosting the required microservice for the specified workflow. This involves dynamic translation between virtual and real IP addresses and ports.

The destination port corresponds to the specific microservice being requested (e.g., port 1 refers to the first microservice). However, for the final response from the last microservice back to the IoT device, the non-reserved port 6000 is used. Microservices can receive requests from various IoT devices, with each request having a unique IP address tied to its specific workflow.

Requests within a workflow travel through the network following the routes specified in the scenario file. These routes are customized by the user in the route field within the workflows. If no routes are specified, the shortest path algorithm is used to determine them. To handle the routing of each request, the switches maintain both a virtual table and a real table (see Fig. 3) for switching different types of traffic.

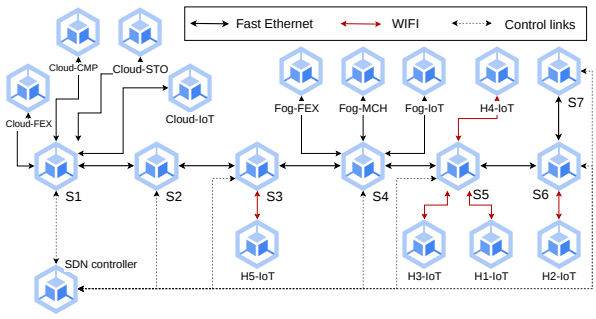


Fig. 4. Scenario used for testing.

The real routing tables are calculated by the EFCC parser using the shortest path algorithm, taking into account the destination IP address and outgoing port. The virtual routing tables are generated based on the construction of virtual requests and the customized routes each must follow. Virtual packets will follow the routes stored in the controller's virtual tables and will be directed through the final switch before reaching the host where the microservice is deployed. The controller translates the virtual packet's address to a real address, so the microservice only processes the real traffic.

Fig. 3 illustrates a sample topology featuring Docker containers and a workflow representation (highlighted in blue with various steps). In this scenario, host *H2* needs to request the microservice *MS2*, deployed on the *Fog* node, but it does not know the microservice's real IP address. The IoT device's request uses a virtual IP address and port: 254.0.0.1, port 1. The controller will automatically redirect the request according to the custom route (or the shortest path), which will be integrated into the virtual tables of each SDN switch (step 2). Upon reaching the final switch *S2* in the path, the controller installs a rule in its flow table to translate the packet's destination IP from a virtual address (e.g., 254.0.0.1, port 1) to a real address (e.g., 8.0.10.11, port 80). After this translation, the virtual IP becomes the real destination IP of the host where the *MS2* microservice is deployed. Consequently, the *MS2* microservice is unaware that the request is ultimately intended for the *MS3* microservice. In step 4, the virtual request travels to the first network node and, upon reaching the final node, the necessary translation is performed (prior to step 5). This workflow requires a response from the *MS3* microservice back to the IoT device, involving virtual request handling in steps 6, 7, and 8. The translation for this response is done at node *S4*.

3) POX component implementation: The custom POX component is implemented via a Python script used to build the controller's Docker image. This image is then assigned to the controller container in the *lab.conf* file. The controller container will run the POX component alongside the network data generated by the EFCC parser. The necessary data for routing and ensuring network connectivity is stored in the *data.json* file within the controller directory. This file contains detailed information on network switches, IP addresses for each container within a host, gateways for each container, user-defined routes, and the routing tables for all switches.

TABLE I
WORKFLOWS TESTED IN THE SCENARIO.

ID	Starter	First microservice	Second microservice	Third microservice
Cloud requests				
1	H1 - IoT	FEX	CMP	STO
3	H3 - IoT	FEX	CMP	STO
5	H5 - IoT	FEX	CMP	STO
7	Fog - IoT	FEX	CMP	STO
9	H2 - IoT	FEX	CMP	STO
11	H4 - IoT	FEX	CMP	STO
13	H1 - IoT	FEX	CMP	STO
15	H3 - IoT	FEX	CMP	STO
Fog requests				
ID	Starter	First microservice	Second microservice	Third microservice
2	H2 - IoT	FEX	MCH	-
4	H4 - IoT	FEX	MCH	-
6	Cloud - IoT	FEX	MCH	-
8	H1 - IoT	FEX	MCH	-
10	H3 - IoT	FEX	MCH	-
12	H5 - IoT	FEX	MCH	-
14	H2 - IoT	FEX	MCH	-

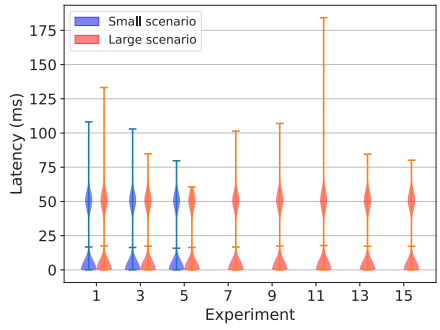
E. IP network support

Although the proposed framework focuses on SDN-based emulated testbeds, it is possible to perform the deployment over an IP network. In order to do this, it must be specified in the *network* field of the scenario file. Traditional IP and SDN paradigms have a number of differences that affect the behaviour of EFCC. IP networks do not leverage controllers, so it is not possible to implement the service discovery functionality. Instead, distributed routing (RIP, OSPF) is considered, implemented using Quagga, and automatically configured through the parser. The code for EFCC, is available as a Git repository [1].

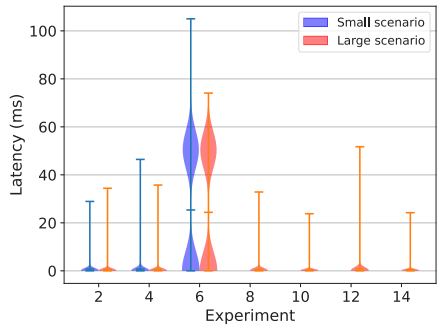
IV. EXPERIMENTAL EVALUATION

This section shows the functioning of EFCC over a realistic smart city use case based on the application in [15]. The application performs automated facial recognition to control facility access, registering authorized users and distinguishing them from intruders. To do so, the application comprises a total of four different microservice types. The first one, noted as *FEX* (*Feature EXtraction*), takes as input the video gathered by an IoT device, leveraging computer vision models to detect faces in the video, segment them, and extract the relevant features for facial recognition, outputting them [15]. The next microservice, labeled *MCH* (*MatCHing*), takes the extracted features as input, and compares them to the features of known users, outputting a boolean that grants or denies access to the facility [15]. These features are saved using the third microservice, *STO* (*STOrage*) [15]. However, due to the high dimensionality of the features, it is necessary to first compress them, a functionality provided by the *CMP* (*CoMPression*) microservice [15]. Therefore, two types of workflows exist, one for registering new users that should be granted access (FEX-CMP-STO), and one for checking whether a user can access the facility (FEX-MCH). This MSA is therefore deployed, with the aim of evaluating network performance and service discovery functionality in EFCC. The considered topology is represented in Fig. 4, which is composed of 7 SDN switches, 7 hosts, and an SDN controller.

[1]https://bitbucket.org/spilab/efcc/src/master/

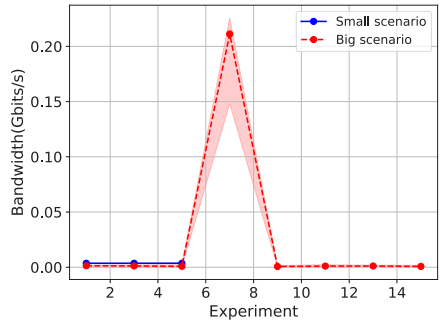


(a) Cloud requests

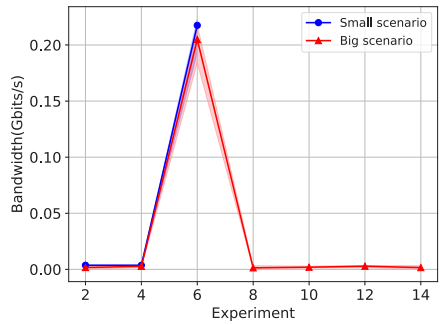


(b) Fog requests

Fig. 5. Average latency per experiment



(a) Cloud requests



(b) Fog requests

Fig. 6. Average bandwidth per experiment

A. *Performance experiments*

For this scenario, we examined two different scenario files. The first file represents a low-load scenario with only 6 workflows. In this setup, requests are sent from all IoT devices deployed across various hosts to the *Fog* and *Cloud* nodes. The fog layer, based on the OpenFog Reference Architecture (OFRA), serves as a processing layer close to the IoT devices within the network. Conversely, the cloud layer functions as a remote processing layer managed by an external provider. Therefore, it has been included in a separate test, as detailed in Table I. The second scenario file expands the number of workflows to assess scalability and its impact on QoS. It includes a total of 15 workflows requested by IoT devices. In addition to the 6 workflows from the first scenario, 9 additional workflows are introduced. These new workflows involve all IoT devices making requests to every microservice deployed across both *Fog* and *Cloud* nodes.

Table I shows the information of each of the IoT requests, including their ID, the starter node, and the workflow (i.e., chain of microservices). A differentiation of *Cloud* requests and *Fog* requests has been made due to their difference in the number of microservices per workflow. For each of the experiments, ten executions have been carried out and the average of the results obtained for both latency and bandwidth metrics was calculated.

After running the EFCC parser with the two scenario files, the results are presented in Figs. 5 and 6, which display the average latency and bandwidth per IoT request. Fig. 5a illustrates the results from workflows arriving at the cloud node, with the Ping tool used for this experiment. It is noted that for the first three experiments, which are common to both scenarios, the smaller scenario consistently exhibits lower latency (on average) compared to the larger scenario due to minor network congestion, specifically between 0.5 and 0.8 seconds less. Additionally, this figure shows higher latency values than those in Fig. 5b, which represents latency for requests made to the *Fog* node. This discrepancy arises because the *Cloud* node has a higher latency in its network access link (refer to Fig. 4), affecting the latency of all requests received.

Fig. 5b illustrates the latency of requests reaching the microservices deployed in the *Fog*. On average, results generally range between 0 ms and 1 ms, with the exception of experiment 6, which shows an outlier. In this case, the IoT device making the request is located within the *Fog* node. The difference in this experiment is due to the IoT device being connected via a Fast Ethernet link, which offers higher capacity and lower latency compared to the Wi-Fi links used by the other hosts (see Fig. 4).

After analyzing latency, Fig. 6a presents the results from the experiment measuring the bandwidth of requests arriving at the *Cloud* node. This measurement was conducted using the `iperf` tool. The results for the first three workflows indicate that bandwidth is higher in the smaller scenario with lower traffic volume. Similarly to Fig. 5b, experiment 7 shows a request from the Fog node to the *Cloud* node, where the access link is Fast Ethernet, resulting in higher bandwidth.

The final result, shown in Fig. 6b, presents the bandwidth measurements for the *Fog* node. These results are comparable to those obtained for the *Cloud* node (refer to Fig. 6a).

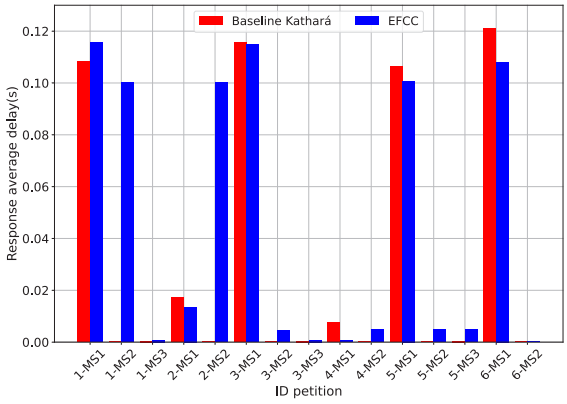


Fig. 7. Average response time in the service discovery experiments.

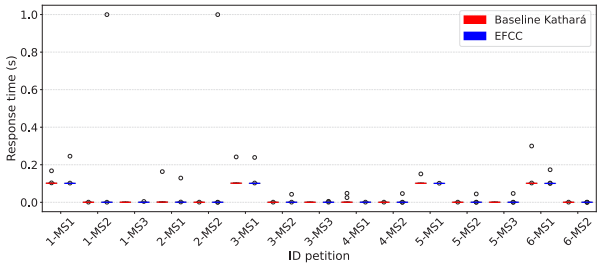


Fig. 8. Box plot representing the response times for the workflows in the service discovery experiments.

TABLE II
IMPACT OF SCENARIO SCALE ON MEMORY USAGE AND DEPLOYMENT TIME

Nodes	Workflows	Memory usage (MB)	Deployment time (s)
14	6	241	10
14	16	252	11
50	35	431	16
100	70	858	30

The lower-loaded scenario generally exhibits higher bandwidth than the larger one. Notably, experiment 6 displays an outlier due to the *Cloud* node sending requests to the *Fog* node via a Fast Ethernet link, which affects the bandwidth measurements.

B. Service discovery experiments

The service discovery implementation was tested using two scenarios: i) the same as before, using EFCC (Kathará + custom POX) with its virtual routing that performs service discovery (virtual scenario), and ii) a baseline scenario that combines Kathará with Faucet (real scenario). This analysis aims to evaluate the performance of virtual routing for service discovery and compare it with traditional traffic in terms of response latencies for various packets. The `tcping` tool was used for these tests, as service discovery operates at the transport layer. For each request, 10 samples were collected to derive statistical metrics. Each test corresponds to a request within a workflow, and the service discovery tests were labelled according to the workflow identifier and the microservices of the first 6 workflows listed in Table I.

Fig. 7 displays a bar chart illustrating the average time taken for each request to reach the target microservice and receive a response. The blue bars indicate the time required in the EFCC scenario, while the red bars represent the results from the baseline scenario. In the baseline scenario, microservices belonging to the same workflow are hosted on the same machine (*Cloud* or *Fog*). Consequently, the first request traverses the entire network. The chart reveals that the highest latencies are observed in the first microservice of each workflow. In two instances (workflow *2-MS2* and workflow *4-MS2*), this pattern does not hold, where the average time for service discovery performed on the host itself exceeds the average service discovery time from the IoT device on the same target node (both cases being *Fog*). This anomaly is likely due to an outlier among the 10 samples, which skews the mean result. These outliers are illustrated in Fig. 8, which features a box plot with very small boxes, indicating low variability in the results. Additionally, there is a significant gap with the outliers. Overall, the medians for the baseline (red line) and EFCC (blue line) tests for the same request are nearly identical (as shown on the graph), demonstrating that the service discovery process has been implemented with minimal performance degradation compared to traditional traffic.

C. Scalability tests of the deployment process.

In this section, we analyze both the computational cost in terms of time associated with emulating scenarios of different sizes using Kathará and the memory consumption of the emulation after deployment, thereby evaluating the scalability of the proposed framework. To this end, experiments are conducted on four scenarios: the two presented in the previous sections, consisting of 14 nodes with 6 and 16 workflows, respectively, and two additional scenarios. The first includes 50 nodes and 35 workflows, while the second comprises 100 nodes and 70 workflows.

Table II shows the results of completing the emulation for each scenario. As observed, both memory usage and deployment time increase with the scale of the scenario. However, the results suggest that the number of workflows does not significantly impact these metrics. Instead, the number of nodes appears to be the dominant factor, strongly influencing both memory consumption and deployment time. Overall, the growth remains moderate, indicating that the proposed framework is able to efficiently handle larger network sizes and workflow demands, thus demonstrating good scalability.

V. CONCLUSIONS

In this work, we propose the EFCC framework to assist researchers in evaluating their deployment models within SDN-Fog environments, where IoT applications following the MSA paradigm are implemented. Additionally, a service discovery mechanism was developed at the network level to automate the routing of requests from IoT devices. As there are no existing tools that integrate the three-layered approach (covering application, computing, and network dimensions), EFCC aims to support researchers and network operators in making informed decisions regarding network, computing, and application deployments while ensuring or optimizing QoS.

Furthermore, no known service discovery tools exist for SDN scenarios. Evaluation in a realistic smart city scenario demonstrates that EFCC can be extended to various scenarios and deployments needed by the Computing Continuum research community and for service discovery purposes.

ACKNOWLEDGEMENTS

This work has been co-financed 85% by the European Union, the European Regional Development Fund and the Regional Government of Extremadura. Managing Authority: Ministry of Finance. Project File Number: IB24102. Grant File Number: GR24099. Projects PID2024-155693NB-C41, 0289_SER65_PLUS_6_P and grant PTQ2024-013825 funded by MICIU/AEI/10.13039/50100011033. This work was supported by the Valhondo Calaff Institution.

REFERENCES

[1] "Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper." [Online]. Available: https://shorturl.at/Xupns

[2] R. Pradhan and A. K. Dash, "An overview of microservices," in *ICDSMLA 2019*, A. Kumar, M. Paprzycki, and V. K. Gunjan, Eds. Singapore: Springer Singapore, 2020, pp. 620–625, DOI: 10.1007/978-981-15-1420-3_66.

[3] J. L. Herrera, J. Galán-Jimnez, J. Garcia-Alonso, J. Berrocal, and J. M. Murillo, "Joint optimization of response time and deployment cost in next-gen iot applications," *IEEE Internet of Things Journal*, vol. 10, no. 5, pp. 3968–3981, 2023, DOI: 10.1109/JIOT.2022.3165646.

[4] S. Dustdar, V. C. Pujol, and P. K. Donta, "On distributed com- puting continuum systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 4092–4105, 2023, DOI: 10.1109/TKDE.2022.3142856.

[5] J. L. Herrera, J. Galán-Jiménez, P. Bellavista, L. Foschini, J. Garcia-Alonso, J. M. Murillo, and J. Berrocal, "Optimal deployment of fog nodes, microservices and sdn controllers in time-sensitive iot scenarios," in *2021 IEEE Global Communications Conference (GLOBECOM)*, 2021, pp. 1–6, DOI: 10.1109/GLOBECOM46510.2021.9685995.

[6] J. L. Herrera, J. Galán-Jiménez, J. Berrocal, and J. M. Murillo, "Optimizing the response time in sdn-fog environments for time-strict iot applications," *IEEE Internet of Things Journal*, vol. 8, no. 23, pp. 17 172–17 185, 2021, DOI: 10.1109/JIOT.2021.3077992.

[7] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 473–478, sep 2012, DOI: 10.1145/2377677.2377767.

[8] V. Kochar and A. Sarkar, "Real time resource allocation on a dynamic two level symbiotic fog architecture," in *2016 Sixth International Symposium on Embedded Computing and System Design (ISED)*, 2016, pp. 49–55, DOI: 10.1109/ISED.2016.7977053.

[9] T. Goyal, A. Singh, and A. Agrawal, "Cloudsim: simulator for cloud computing infrastructure and modeling," *Procedia Engineering*, vol. 38, pp. 3566–3572, 2012, DOI: 10.1016/j.proeng.2012.06.412.

[10] I. Lera, C. Guerrero, and C. Juiz, "Yafs: A simulator for iot scenarios in fog computing," *IEEE Access*, vol. 7, pp. 91 745–91 758, 2019, DOI: 10.1109/ACCESS.2019.2927895.

[11] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *Journal of Network and Computer Applications*, vol. 75, pp. 138–155, 2016, DOI: 10.1016/j.jnca.2016.09.001.

[12] G. Bonofiglio, V. Iovinella, G. Lospoto, and G. Di Battista, "Kathará: A container-based framework for implementing network function virtualization and software defined networks," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–9, DOI: 10.1109/NOMS.2018.8406267.

[13] S. Laso, J. Berrocal, P. Fernández, A. Ruiz-Cortés, and J. M. Murillo, "Perses: A framework for the continuous evaluation of the qos of distributed mobile applications," *Pervasive and Mobile Computing*, vol. 84, p. 101 627, 2022, DOI: 10.1016/j.pmcj.2022.101627.

[14] A. García-Fernández, S. Laso, J. A. Parejo, J. Berrocal, A. Ruiz-Cortés, and J. M. Murillo, "Towards effective saas pricing design: A case study of ccsim," in *Service-Oriented Computing – ICSOC 2024 Workshops*, S. Kallel, C. Raibulet, I. Bouassida Rodriguez, N. Faci, A. Bennaceur, S. Cheikhrouhou, L. Ben Ayed, M. Sellami, E. Y. Nakagawa, and R. Ben Halima, Eds. Singapore: Springer Nature Singapore, 2026, pp. 157–168, DOI: 10.1007/978-981-96-7238-7_13.

[15] S. Wang, M. Zafer, and K. K. Leung, "Online placement of multi-component applications in edge computing environments," *IEEE Access*, vol. 5, pp. 2514–2533, 2017, DOI: 10.1109/ACCESS.2017.2665971.

José Gómez-delaHiz (Student Member, IEEE) is a Ph.D. student at the University of Extremadura, Spain, where he has been awarded with a Valhondo predoctoral fellowship. His current research focuses on UAV-based networking, Green networking, and Intent-Based Networking. Contact him at jagomezdh@unex.es

Juan Luis Herrera is a MSCA postdoctoral fellow in the Distributed Systems Group at TU Wien. He received a Ph.D. in Computer Science from the University of Extremadura in 2023. His main research interests include IoT, CC, service management, and sustainability.

Sergio Laso received the Ph.D. in Computer Science from the University of Extremadura (Spain) in 2023. He is currently working at the Department of Computer Systems and Telematics Engineering of the University of Extremadura (Spain) and as Industrial Ph.D at Gloin S.L. His research interests focus on the distributed systems and computing continuum architectures. Contact him at slasom@unex.es

Javier Berrocal (Member, IEEE) received the Ph.D. degree in Information Technologies for the University of Extremadura in 2014, where he is currently an associate professor. His main research interests are software architectures, pervasive systems, and computing continuum. Contact him at jberolm@unex.es

Jaime Galán Jiménez (Member, IEEE) received the Ph.D. in Information Technologies from the University of Extremadura in 2014, where he is currently an associate professor. His research interests are network digital twins, AI/ML for network management, and UAV-based networks. Contact him at jaime@unex.es.