

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

EFCC: a flexible Emulation Framework to evaluate network, computing and application deployments in the Cloud Continuum

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:

León, L.J.M., Herrera, J.L., Berrocal, J., Galán-Jiménez, J. (2023). EFCC: a flexible Emulation Framework to evaluate network, computing and application deployments in the Cloud Continuum [10.1109/iscc58397.2023.10218062].

Availability:

This version is available at: <https://hdl.handle.net/11585/959563> since: 2024-02-20

Published:

DOI: <http://doi.org/10.1109/iscc58397.2023.10218062>

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (<https://cris.unibo.it/>).
When citing, please refer to the published version.

(Article begins on next page)

EFCC: a flexible Emulation Framework to evaluate network, computing and application deployments in the Cloud Continuum

Luis Jesús Martín León, Juan Luis Herrera, Javier Berrocal and Jaime Galán-Jiménez
Department of Computer Systems and Telematics Engineering, University of Extremadura, Spain.
[luismml, jlherrera, jberrocal, jaime]@unex.es

Abstract—In recent years, the number of devices connected to the Internet (and hence the data traffic) has significantly increased. The adoption of the Internet of Things paradigm, the use of the MicroServices Architecture for applications and the possibility of deploying such applications at different layers (fog, edge, cloud), makes the selection of an appropriate deployment a critical task for network operators and developers. In this paper, an emulation framework is proposed to allow them make a decision for the network, computing and application deployment in the cloud continuum, while satisfying the required Quality of Service. The framework is compatible both for IP and SDN network paradigms and is extensible to different types of scenarios thanks to its approach based on Docker containers. The evaluation over a realistic network scenario shows that it is extensible to any scenario and deployment required by the research community working on the cloud continuum.

Index Terms—Cloud continuum, SDN, framework, Fog, IoT

I. INTRODUCTION

The Internet of Things (IoT) is an emerging technology with a high significance related with technical, social and economic aspects. Nowadays, it is common to use IoT devices in different environments to perform daily tasks through multiple communication models, e.g., smart plugs, fridge sensors, GPS, parking and agriculture sensors, etc. Moreover, it also introduces new concepts about energy efficiency and security in smart cities or smart homes.

IoT devices usually communicate with services, which are pieces of software that are distributed throughout different computing devices. The MicroService Architecture (MSA) allows to split a monolithic application into multiple small applications with independent responsibilities. The set of ordered microservices belonging to an application is known as workflow and user requests are associated a single type of workflow.

The increase in the number of IoT devices presents a direct impact over the traffic that is flowing through the Internet. The number of devices connected to IP networks is approximately three times the world's population in 2023, about 29.3 billion networked devices [1]. Along with new applications that require strict Quality of Service (QoS) levels such as, e.g., holoportation, metaverse, etc., the management of large volumes of data flowing through the networks is a challenge for network operators.

In an IoT-based network scenario, three layers must be considered to optimize the QoS: i) the network layer, in which

network resources are managed to optimize the network QoS; ii) the computing layer, where computing devices provide computing resources to allow IoT applications run on top of them; and iii) the application and services layer, for which the QoS of applications is aimed to be optimized.

The application and service layer refers to IoT applications which are designed following the MSA pattern. MSA applications can be differentiated through division into small loosely-coupled microservices [2]. If simple tasks collaborate, more complex functionalities can be handled. Moreover, distributed implementation provides higher flexibility. Application logic and device capacity directly affect the application performance, in other words, latency and response time. In order to optimize the application QoS, the problem that aims at optimally placing the set of microservices in an infrastructure is known in the literature as the Decentralized Computation Distribution Problem (DCDP) [3].

The computing layer focuses on a smart service distribution. Commonly, cloud computing was used to deploy IoT applications. However, cloud servers tend to be far away from IoT devices. Thus, it is convenient to move microservices to fog computing nodes because they are closer to users and the applications response time is lower. As result, fog nodes location is crucial to optimize the QoS. Such problem is known as Fog Node Placement Problem (FNPP) [4].

Finally, the network layer is in charge of communications. A key aspect in MSA-based IoT application environments is the scalability and flexibility of the network. Since microservices are replicated in different network nodes, service discovery must be carried out. The emergence of enablers such as Software-Defined Networks (SDN) and Network Function Virtualization (NFV) perfectly fit with the idea of decentralising IoT applications using the MSA paradigm. SDN separates control and data planes, whose correct interaction is crucial for achieving the desired network QoS. Thus, the Controller Placement Problem (CPP) [5] is a key problem to be solved in this type of scenarios, putting the SDN controller wherever the QoS from the switches is best.

Generally, companies tend to reduce the deployment cost and improve the network performance. To achieve such goals, a multi-objective deployment can be carried out [4] by solving the CPP, DCDP and FNPP problems. In this paper, an emulation framework is provided to emulate adhoc solutions from

researchers in terms of CPP, DCDP, FNPP and any combination of them. So far, there are no tools that can combine a three-layered scenario considering application, computing and network dimensions [6], [7], [8] the main goal of this work is to propose a framework that would help researchers and network operators to make a decision for the network, computing and application deployment, while satisfying or optimizing the QoS. Specifically, the main contributions of this work are the following ones:

- The proposal of a framework to deploy IoT applications based on microservices in a Fog-SDN based scenario. There is the possibility to include different type of IoT applications composed by different kind of microservices.
- The evaluation of the proposed framework over diverse realistic IoT scenarios and use cases.

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

II. EFCC FRAMEWORK

In this section, the proposed framework, namely Emulation Framework to evaluate network, computing and application deployments in the Cloud Continuum (EFCC), is described according to its different modules. EFCC is composed of 3 different modules: i) *Dadosim* file (Sec. II-A), ii) EFCC parser (Sec. II-B), and iii) Kathará laboratory (Sec. II-C) with the corresponding Docker containers.

A. Dadosim file

Dadosim file is a user personalized file that stores all the data that is required as input by the framework. *Dadosim* format is similar to JSON format and data is organized in blocks as in Fig. 1a, which represents the scenario of Fig. 1b. The different blocks are described next:

- **Network:** the framework supports two different types of computer networks: IP and SDN. Depending on the type of network, the behaviour of framework will be different.
- **Microservices:** list of different microservices to be used by the applications in the framework. It is important to differentiate microservices through their identifier, and to indicate the URL of the Docker image which the microservice is going to use.
- **Hosts:** in order to create the network topology, hosts must be defined with an identifier and specific requirements such as power (in CPU percentage) and memory (in bits). Moreover, information about Capex (Capital Expenditures), and Opex (Operating Expense) can also be specified (they are not mandatory). Finally, a list with the set of microservices that will be deployed at each host is defined (it can be also empty).
- **Switches:** as in the case of hosts, switches are defined by an identifier. When the attribute "controller" matches with the identifier and the "deployed" attribute is true, then this switch is directly connected to the SDN controller. Furthermore, this block related to switches can have

optional metrics for the Capex and Opex. The route field indicates the deployment path of the controller associated with that switch. If it is not indicated, a single default controller will be used.

- **Workflows:** a list with the different types of workflows is required. Each workflow represents the information associated to an IoT request by a specific user. The first attribute is "chain", which represents a list with the microservices that are going to take action in this workflow. This chain is an ordered set of microservices that must be executed according to the application requirements. The IoT device will be deployed in the host that it is indicated in the "starter" attribute. Besides, the docker image for the IoT application is defined in the "iot_image" attribute. If the last microservice requires to send back a response to the IoT device that requested the application, the "response" attribute would be true, whereas if the response is not necessary this attribute is false. The last attribute for the *Workflows* block is the chain deployment, which includes the information of each microservice that is required by the workflow (the same microservices indicated in the first attribute). Every microservice has its node deployment (fog or cloud), the port at which the microservice will listen the incoming requests and a list of hosts and switches with the route or path that the request should follow to reach the destination. If the route is not specified, it is automatically calculated by the framework using the shortest path according to Dijkstra algorithm.
- **Parameters:** this block includes additional information as the control message size to be able to, e.g., assess the overhead in the network. It is open to new potential parameters that the model to be evaluated by the user could require.
- **Links:** to finally build the network topology, the set of links with their source and destination nodes, as well as their latency (in milliseconds) and capacity (in Bytes) must be specified.

Some capacities metrics are expendable, that is to say, it is possible to omit this attributes, for example, capex and opex metrics in links, switches and hosts. The user must indicate the file with ".dadosim" extension to execute the Parser program that is described in the next section II-B.

B. EFCC parser

Once the input data required by the framework is described by means of the *Dadosim* file, a parser has been developed with the goal of creating Docker containers [9] and connect them to represent the Fog-SDN scenario with IoT applications based on the MSA paradigm.

1) *Containers:* The elements of the previously explained *Dadosim* file that have a direct translation onto a Docker container are the network elements, i.e., switches (both for IP and SDN networks) and the SDN controller if the scenario is an SDN one. Moreover, a container is created for each Cloud or Fog node that has a deployed microservice, whose



Figure 1. Dadosim structure

characteristics are specified in the *Dadosim* file (including the microservice Docker image). Docker does not allow to use more than one image within a container, which is an important constraint. It must have as many containers as microservices. In addition, microservices will receive requests from the IoT devices deployed in the end hosts, and each of these IoT devices also has a single Docker container associated with it. For instance, in Fig. 2 Fog node has two microservices deployed, which is translated into two Fog containers. It is possible to deploy a microservice and an IoT device in the same host, as the Docker containers are deployed in the same way, that is to say, separated. If the user aims at effectively limiting network metrics (bandwidth and delay) it is necessary to have the *iproute* package installed within the Docker images due to the fact that the *tc* command needs such library. Otherwise, QoS values could not be gathered.

2) *Workflows*: Workflows translate requests coming from IoT devices into microservices (deployed in Fog or Cloud nodes). As previously explained, each workflow is composed by a chain of microservices, which have functionalities that are executed following an specific order. The objective is to satisfy the request made by the IoT device defined as source node. In Fig. 2, a representation of a workflow is shown in red colour. This workflow is requested by the IoT device of host 2 (*H2_IoT_device2*) and requires microservices 2 and 3

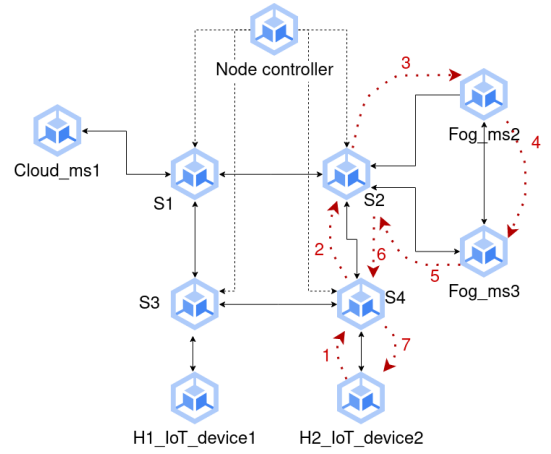


Figure 2. Docker container view

located in the Fog node.

In the deployment chain of each microservice the user can specify the port where it will be deployed within the host (Fog or Cloud) and the route that the request will follow until it reaches the host. The routing information will be reflected in a JSON file as input for the configuration of our controller (explained later in Sec. II-D). If any microservice does not have such a route, Dijkstra algorithm will be executed to assess the shortest path between the IoT device and the destination host, where the microservice is deployed.

3) *Creation and association of links and domains*: Once all the containers are created and ready, the *Dadosim* links are registered. These links are in charge of connecting switches and hosts to build the network topology. However, we find a limitation due to the use of Docker technology. It will be necessary to create multiple virtual links, as many as containers correspond to a host, to replicate a real link in several virtual links, due to the above mentioned restriction (Docker only allows to store one microservice or IoT device per container, Sec. II-B1). Real links are understood as those links that are explicitly specified in the *Dadosim* input file. Multiple virtual links make it possible to connect different containers and assume the behavior of one unit. Since the creation of the links is one of the most complex parts of the framework, its main structure is presented in the form of pseudocode.

Alg. 1 allows to achieve network connectivity by creating and associating different domains with the containers that will act as switches, IoT devices and microservices. First of all, it is necessary to know if there is a microservice deployed on a host or an IoT device, since it is possible to deploy different microservices on the same host (lines 5-15). This number is stored in order to create as many domains as necessary (line 7). Once the domains have been created, it is necessary to associate them with the link information among a switch with the host IoT microservice/device or among two different switches (lines 9,10,15). Next, the source and destination nodes of each link is recorded according to its type (lines

Algorithm 1 Pseudo code of links creation

```
1: for all links in Dadosim do
2:   if switch to switch link then
3:     create and register the link
4:   else
5:     if (iotcounter + microservicecounter) > 1 then
6:       if !domain then
7:         Create domain
8:       end if
9:       link ← domain
10:      linklist ← link
11:     else
12:       if !domain then
13:         Create domain
14:       end if
15:       link ← domain
16:     end if
17:     if switch to host link then
18:       register link in source node
19:       register link in destination node
20:     else if host to switch link then
21:       register link in destination node
22:       register link in source node
23:     end if
24:     Add network gateway
25:   end if
26: end for
```

17-23). Finally, the microservice/IoT device gateways of the hosts will be added (line 24).

4) *Domain features*: Domains are considered as class C networks whose network ID (by default) is built by concatenating a first part (by default 8), a second part (by default 0), a third part (by default 0) and the fourth and last part corresponds to the host identifier. A new network ID is created increasing by one unit the default address from the third part to the first part when each of them fill up its range (0-255). When we need to add a host to a network, its network ID is used in such a way that the next available IP address is considered.

C. Kathará laboratory and Docker containers

Kathará is a network virtualization tool that allows to create and configure complex network topologies for testing and emulation using containers [10]. Once we have collected all the input information (*Dadosim* file) for the creation of the environment in Kathará, we will proceed to write the configuration files needed to run the scenario. To create and configure containers to be used in the topology, the configuration file ".startup" is used by Kathará, which allows us to define the actions to be taken when each container is started. Depending on the type of node that is considered (IoT devices, microservices or switches), the configuration is different.

- **IoT devices and microservices**: IoT devices and microservices containers will configure their network interfaces using the *ifconfig* command, indicating the IP address of each interface, the netmask and the broadcast addresses. The *tc* command will be used to limit the bandwidth and delay of the corresponding interfaces. Finally, the default gateway is registered for the host.
- **Switches**: the configuration file of a switch is more complex than the previous ones since it will include a set of Open vSwitch configuration commands. Such

commands are used to connect the interface of a switch with the virtual ports that are necessary in each case. The network interfaces will receive their IP address using the *ifconfig* command and their delay and bandwidth will be limited using the *tc* command.

Kathará uses the "lab.conf" file to define the configuration of the network topology to be created and emulated. This file describes the Docker image per node, the type of shell, the node processing capacity (for microservices and IoT devices), the node memory capacity (for microservices or IoT devices) and the association of its interfaces with its corresponding domain. Additionally, a process has been designed to detect problems in the deployment of the scenario in Kathará. When all nodes finish their configuration, they create a text file in the shared folder between the system and the virtual scenario where they will write a message to indicate that they have been successfully deployed. After starting the scenario in Kathará we will execute a Python script that will check that all containers have been deployed correctly and in case of any error it will restart the environment.

D. POX controller

The proposed framework is devised to follow the SDN paradigm and therefore a controller is required. For this reason, POX controller [11] has been considered to act as SDN controller thanks to its flexibility and degree of adaptation for our case uses.

1) *POX component behavior*: In the following, the basic network functions required to ensure connectivity between IoT devices and microservices are explained. The controller will be responsible of installing the necessary flow rules in the SDN switches. The communication among switches and the controller is provided by the OpenFlow (OF) protocol and will allow traffic flows to be scheduled in the SDN network. Routing decisions are responsibility of the user of the framework, as already explained in Sec. II-A. In case no routing is specified, the shortest path rule is considered.

2) *POX component implementation*: The implementation of the custom POX component is done through a file written in Python that will be used to build the Docker image. Subsequently, this image will be assigned to the controller container in the "lab.conf" file. The controller container will use the POX component along with the network data generated by the EFCC Parser. The data needed to perform routing and ensure network connectivity is stored in the *data.json* file, which is located inside the controller directory. This file will have all the information related to the network switches, the IP addresses of each container that are part of a host, the gateways for each container, the user-customized routes and the routing tables of all switches in the network. The controller will use this information in its various modules to manage all network events and to be able to make the right decisions at all times.

E. IP networks

Although the proposed framework focuses on the emulation over SDN networks, it is possible to perform the deployment

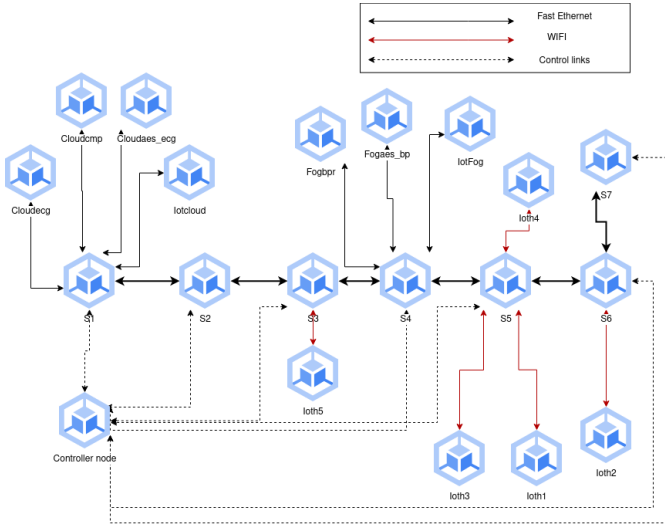


Figure 3. Scenario for the tests

over an IP network. In order to do this, it must be specified in the "network" field of the *Dadosim* file, as described in Sec. II-A. Traditional IP and SDN paradigms have a number of differences that affect the behavior of our proposed framework. At first, no controller is required in an IP network. Instead, distributed routing (RIP, OSPF) is considered by means of Quagga tools [12]. The framework code is available in the Bitbucket repository ¹.

III. EXPERIMENTAL RESULTS

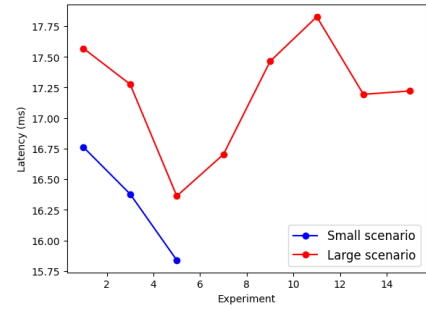
This section shows the functioning of EFCC Framework over a realistic use case. The considered topology is represented in Fig. 3, which is composed of 7 SDN switches, 7 hosts and a SDN controller. For the same scenario, we have considered two different *Dadosim* files: The first file corresponds to a low-loaded scenario in which only 6 workflows are considered. Requests are made from all IoT devices deployed in the different hosts toward the Fog and Cloud nodes. In this case, each host has deployed the microservices on which it makes requests, either in Cloud or in Fog. The second *Dadosim* file increases the number of workflows to evaluate the scalability over the resulting QoS. A total of 15 workflows are requested by IoT devices. Apart from the same 6 workflows of the first scenario, 9 additional workflows are created. With the new workflows we consider all IoT devices to make requests to all microservices deployed in 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

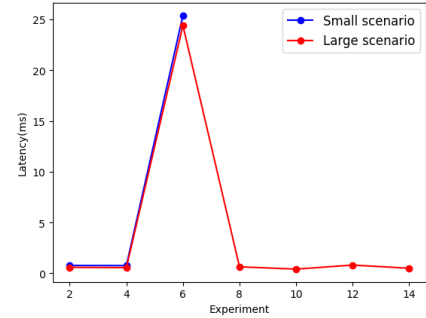
Cloud requests				
ID	Starter	Ms 1	Ms 2	Ms 3
1	h1	ecg	cmp	aes_ecg
3	h3	ecg	cmp	aes_ecg
5	h5	ecg	cmp	aes_ecg
7	fog	ecg	cmp	aes_ecg
9	h2	ecg	cmp	aes_ecg
11	h4	ecg	cmp	aes_ecg
13	h1	ecg	cmp	aes_ecg
15	h3	ecg	cmp	aes_ecg

Fog requests				
ID	Starter	Ms 1	Ms 2	Ms 3
2	h2	bpr	aes_bp	-
4	h4	bpr	aes_bp	-
6	cloud	bpr	aes_bp	-
8	h1	bpr	aes_bp	-
10	h3	bpr	aes_bp	-
12	h5	bpr	aes_bp	-
14	h2	bpr	aes_bp	-

Table I
WORKFLOWS



(a) Cloud requests



(b) Fog requests

Figure 4. Average latency per experiment

metrics was calculated. The traffic volume has been generated with tools: Iperf (limited in time according to the test) and Ping (limiting the number of packets to 20).

After executing EFCC Parser with the two *Dadosim* files, the obtained results are shown in Figs. 4 and 5, showing the average latency and the average bandwidth per IoT request. Figure 4(a) shows the results obtained from the execution of the workflows arriving at the cloud node. Ping tool has been used to perform this experiment. It is observed that for the first three experiments (common in the two scenarios) the small scenario has always lower latency than the larger scenario due to a slightly network congestion. Moreover, this figure shows higher latency values than those obtained in Fig. 4(b), which

¹<https://bitbucket.org/spilab/efcc/src/master/>

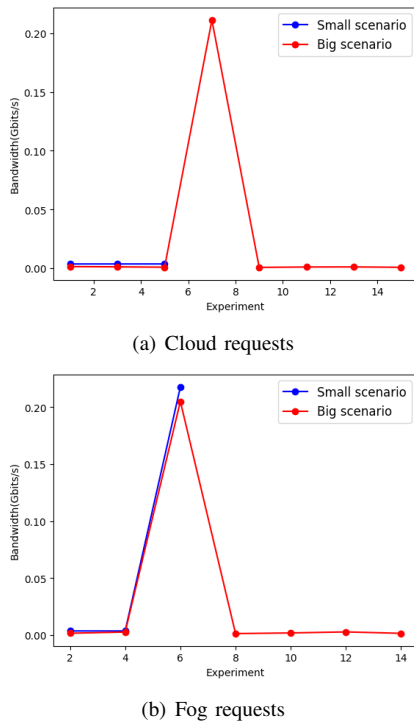


Figure 5. Average bandwidth per experiment

depicts the latency of the requests made toward the Fog node. This behavior is due to the fact that the Cloud node has a higher latency in its network access link (see Fig. 3) and this has an impact on the average of all requests arriving to it.

Figure 4(b) shows the average latency of the requests arriving to the microservices deployed in the Fog. We can notice that the results oscillate between the range of 0 ms and 1 ms, except in experiment 6. In this experiment we have an outlier because the IoT device making the request is located in the Fog node. The difference with the rest of the experiments is that this IoT device is connected to the network through a Fast Ethernet link with higher capacity and lower latency than the WiFi links used by the rest of the hosts (see again Fig. 3).

Once latency has been analyzed, Fig 5(a) shows the results of the experiment performed to measure the bandwidth of the requests arriving to the Cloud node. This experiment has been performed using the Iperf tool. Results obtained for the first three workflows show that the bandwidth is higher in the small scenario where there is a lower traffic volume. On the other hand, we find a similar behavior to the previous figure (Fig. 4(b)), since experiment 7 performs a request from the Fog node to the Cloud node, so that its access link to the network is of type Fast Ethernet. The last outcome represented in Fig. 5(b) shows the results of the experiment performed on the Fog node to measure its bandwidth. These results are very similar to those obtained in the experiments performed on the Cloud node (see Fig. 5(a)). The low-loaded scenario tends to have higher bandwidth than the large one. In this case, experiment 6 is the one that shows an outlier result and it is due to the fact that the Cloud node sends requests towards the Fog node

using a Fast Ethernet link in its network access.

IV. CONCLUSIONS

In this work, the framework EFCC has been proposed to help researchers evaluate their deployment models in SDN-Fog scenarios where IoT applications based on the MSA paradigm are run. Since there are no tools that combine a three-layered scenario considering application, computing and network dimensions, EFCC will help researchers and network operators to make a decision for the network, computing and application deployment, while satisfying or optimizing the QoS. The evaluation over a realistic network scenario shows that it is extensible to any scenario and deployment required by the research community working on the cloud continuum.

ACKNOWLEDGEMENTS

This work was partially funded by the project PID2021-124054OB-C31 and the grant CAS21/00057 (MCI/AEI/FEDER, UE), by the grant PDC2022-133465-I00 and the project TED2021-130913B-I00 funded by MCIN/AEI/10.13039/501100011033 and by the “European Union NextGenerationEU/PRTR”, by the Department of Economy, Science and Digital Agenda of the Government of Extremadura (GR21133), and by the European Regional Development Fund.

REFERENCES

- [1] “Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper.” [Online]. Available: <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>
- [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.
- [3] 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.
- [4] 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.
- [5] B. Heller, R. Sherwood, and N. McKeown, “The controller placement problem,” *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, p. 473–478, sep 2012.
- [6] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, “Fogify: A fog computing emulation framework,” in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, 2020, pp. 42–54.
- [7] A. Coutinho, F. Greve, C. Prazeres, and J. Cardoso, “Fogbed: A rapid-prototyping emulation environment for fog computing,” in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–7.
- [8] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran, “Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures,” in *2017 IEEE Fog World Congress (FWC)*, 2017, pp. 1–6.
- [9] “Docker: Accelerated, Containerized Application Development,” May 2022. [Online]. Available: <https://www.docker.com/>
- [10] 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.
- [11] N. O. X. Repo, “POX,” Apr. 2023, original-date: 2012-02-07T22:18:46Z. [Online]. Available: <https://github.com/noxrepo/pox>
- [12] P. Jakma and D. Lamparter, “Introduction to the quagga routing suite,” *IEEE Network*, vol. 28, no. 2, pp. 42–48, 2014.