

# Monitoring Platform Evolution towards Serverless Computing for 5G and Beyond Systems

Ramon Perez, Priscilla Benedetti, Matteo Pergolesi, Jaime Garcia-Reinoso, Aitor Zabala, Pablo Serrano, Mauro Femminella, Gianluca Reali, Kris Steenhaut, Albert Banchs

**Abstract**—Fifth generation (5G) and beyond systems require flexible and efficient monitoring platforms to guarantee optimal key performance indicators (KPIs) in various scenarios. Their applicability in Edge computing environments requires lightweight monitoring solutions. This work evaluates different candidate technologies to implement a monitoring platform for 5G and beyond systems in these environments. For monitoring data plane technologies, we evaluate different virtualization technologies, including bare metal servers, virtual machines, and orchestrated containers. We show that containers not only offer superior flexibility and deployment agility, but also allow obtaining better throughput and latency. In addition, we explore the suitability of the Function-as-a-Service (FaaS) serverless paradigm for deploying the functions used to manage the monitoring platform. This is motivated by the event oriented nature of those functions, designed to set up the monitoring infrastructure for newly created services. When the FaaS warm start mode is used, the platform gives users the perception of resources that are always available. When a cold start mode is used, containers running the application's modules are automatically destroyed when the application is not in use. Our analysis compares both of them with the standard deployment of microservices. The experimental results show that the cold start mode produces a significant latency increase, along with potential instabilities. For this reason, its usage is not recommended despite the potential savings of computing resources. Conversely, when the warm start mode is used for executing configuration tasks of monitoring infrastructure, it can provide similar execution times to a microservice-based deployment. In addition, the FaaS approach significantly simplifies the code logic in comparison with microservices, reducing lines of code to less than 38%, thus reducing development time. Thus, FaaS in warm start mode represents the best candidate technology to implements such management functions.

**Index Terms**—5G networks, monitoring and data collection, serverless computing, FaaS, performance comparison

Manuscript received on xx yy, 2021. This work has been supported by EC H2020 5GPPP projects 5G-EVE and 5GROWTH under grant agreements No. 815974 and 856709, respectively. The views expressed are those of the authors and do not necessarily represent the projects. The Commission is not responsible for any use that may be made of the information it contains.

M. Pergolesi, and A. Zabala are with Telcaria Ideas, Spain. P. Benedetti, M. Femminella, and G. Reali are with the Department of Engineering, University of Perugia, Italy. M. Femminella, and G. Reali are also with Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), Italy. K. Steenhaut is with Vrije Universiteit Brussel, Department of Electronics and Informatics (ETRO), Belgium. P. Benedetti is also with Vrije Universiteit Brussel, Department of Electronics and Informatics (ETRO), Belgium. R. Perez, P. Serrano, and A. Banchs are with Universidad Carlos III de Madrid, Spain. A. Banchs is also with IMDEA Networks Institute, Spain. J. Garcia-Reinoso is with Department of Automatica, Universidad de Alcalá, Spain. Corresponding author: Mauro Femminella (e-mail: mauro.femminella@unipg.it).

## I. INTRODUCTION

The growth of data-intensive and real-time applications, empowered by 5G and Internet of Things (IoT) devices, has gradually triggered the shift from centralized data centers to distributed computing and storage resources, in order to avoid potential bottlenecks and low throughput caused by services that process increasing data volumes. This architectural change has subsequently fostered the evolution from traditional cloud computing services to more capillary models, such as Edge computing [1] and Fog computing [2]. The boundary between the two technologies is blurred; Edge computing is typically regarded as a service implemented either on the devices to which the user-terminals are attached to or very close to it, as a rack connected via a gateway node to the access network. According to the ETSI vision [3], an Edge computing application is meant to deploy services close enough to the end users in order to save transmission resources and to enable low-latency applications with respect to a mere cloud deployment. Fog Computing entails services taking place in a continuum between the access network, close to end devices, and a central cloud [4].

5G systems and their expected evolution, referred to as Beyond 5G (B5G) systems, are expected to encompass Fog and Edge computing technologies to satisfy the service requirements of new classes of applications requested by many of their potential vertical customers. Such customers could be stakeholders of other business sectors that move to 5G as main transport infrastructure, that require to respect desired key performance indicators (KPIs) while keeping an efficient resource usage. Vertical industries may be interested to use B5G systems not only as transport network, but also as support for deploying their services (or part of them) in data centers hosted in the 5G network itself, including Edge and/or core cloud clusters, as part of a dedicated network slice [5]. Indeed, the increasing volume of data coming from IoT devices is going to be processed in a distributed way on B5G platforms, relying on an Edge/Fog layer in the architecture. Hence, the importance of monitoring services collecting and analyzing data generated by the different elements involved in B5G systems, and generating the KPI values suitable for vertical customers, is clear [6]. These platforms need to be lightweight and flexible to be easily deployable in Edge environments, and should limit the resource allocation for their modules. However, the conventional provisioning of virtual machines (VMs) initially designed for 5G network services may significantly limit the number of concurrent applications and users in an

Edge or Fog layer of the architecture. Indeed, it relies on overprovisioning, large computing overhead, and long startup times. Computing nodes for lightweight deployment have limited resources with respect to a core cloud system. This is true not only for the vertical applications, but also for the whole management infrastructure, which has a significant impact on the overall amount of resources to be deployed. Thus, also monitoring services for 5G systems have to move towards more lightweight deployment paradigms. In this regards, the main candidates are containerized packages [7] running as microservices [8], and serverless computing [9], which we will evaluate in this paper as possible technological alternative for implementing monitoring services in 5G systems.

Deploying applications components in containers as microservices, and orchestrating them with platforms like Kubernetes, facilitates the service management, by providing auto scaling and high availability mechanisms, and increase service agility in 5G systems [10]. Indeed, the virtualization support in Kubernetes and Docker is an enabler for implementing network slicing in 5G, since it provides service differentiation between services offered to different verticals. Serverless computing moves a step forward. In a serverless architecture, computing resources, along with their configuration and management, are dynamically provisioned at runtime — thus the name serverless: It hides server provisioning, maintenance, updates, scaling, and capacity planning from developers and users. Function-as-a-Service (FaaS) model is the primary implementation of serverless computing. It allows application logic, written as stateless functions, to be executed on demand by containerized runtime environments, without pre-allocating resources [11] [12]. FaaS and microservices share similar properties, i.e. the focus on deployment cycle’s flexibility by dynamically handling changing development requirements. Both technologies share a modular architectural pattern, although FaaS is specifically made of atomic functions meant to be stateless and, typically, event-driven. Microservices are kept active even when not required, whereas the FaaS approach allows activating functions on-demand to save resources.

This paper is meant to be a further development of the work presented in [13], in which we compared the performance of several virtualization technologies used to deploy a 5G multi-site and multi-stakeholder monitoring platform. For this preliminary comparison, we used the VM-based platform developed in the 5G EVE project [14], whose monitoring framework allows implementing the distribution and consumption of metrics and KPIs for 5G multi-site platforms [6], [15] through a publish/subscribe approach using *Apache Kafka*<sup>1</sup>, which demonstrated to be effective. Driven by the promising results obtained by using different virtualization technologies in [13] for implementing the communication system in charge of delivering streams of monitoring data to data collectors, in this paper we significantly extend the analysis. Furthermore, the promising initial results with container-based modules motivated us also to investigate the introduction of serverless technologies in the management plane of the

monitoring platform. In particular, the event-triggered nature of its modules enables the transition from microservices to FaaS for all the functions that have to be executed at the setup or teardown of a new slice, in order to configure the relevant monitor infrastructure. Indeed, managing even some modules of the monitoring platform by using serverless functions would greatly simplify the development, reduce the deployment and configuration effort, and enable the effortless adoption of horizontal scaling.

Previous work covered the introduction and first analysis of the 5G monitoring platform, based on a set of VMs [6], as well as an initial performance comparison between different virtualization technologies including recent lightweight technologies, such as Kata Containers and Firecracker [13]. This work significantly advances these results. The two main contributions are as follows.

We significantly extend the performance evaluation presented in [13]. In particular, we focus on the suitability of the considered virtualization technologies for what concerns the evolution towards a lightweight version of the target 5G monitoring platform. We present extensive experimental results comparing the deployment of the platform through bare metal, VM, and orchestrated containers, providing novel results on scalability and transient behavior, not present in [13]. Our measurements are carried out through an experimental testbed. It includes two servers located in the same LAN, one hosting the platform to evaluate, and the other being in charge of generating the workload.

We present a new design of 5G monitoring platforms, compared with [6], by using both microservices and the serverless principles. In more detail, we illustrate how the serverless deployment can be leveraged to realize the functions used to manage event-based features of the monitoring platform, i.e. those in charge to handle Kafka topics. In addition, we evaluate the performance of the new system in terms of execution time and developer effort, by comparing a FaaS deployment, based on both cold and warm start modes, against a microservices-based one. We used the open-source platform *OpenFaaS* [16], orchestrated by *Kubernetes* [17], for the FaaS deployment testbed, whereas microservices are deployed as containers with *Docker* [18] and *runc* as runtime, orchestrated by *Kubernetes* as well. According to the experimental results, we discuss the resulting benefits and drawbacks with respect to the original implementation.

In conclusion, the study illustrated in this paper leads to the proposal of a new design for 5G monitoring systems with respect to the state of the art. Investigating on the performance comparison between different virtualization technologies and deployment paradigms, this work depicts the suitable solutions available to evolve the considered platform towards a more efficient and lightweight version. In fact, our findings suggest to move from a classic VM-based platform to an hybrid one, in which data monitoring is handled through orchestrated containers, whereas the relevant management and configuration functions are implemented through the novel FaaS serverless

<sup>1</sup><https://kafka.apache.org/>

paradigm, configured in warm start mode.

The rest of the paper is organized as follows: Section II presents some background material related 5G monitoring platforms as well as to the main concepts of microservices and serverless computing. Section III presents an overview of the monitoring platform under study, describing its building blocks. It introduces the transformation from the original platform modules to serverless functions exposed as a service, and compares the FaaS modules-based version with the microservices-based one. Section IV presents the analysis of the platform, consisting of a performance evaluation of different deployment techniques for the considered platform, followed by the evaluation of a serverless deployment of the platform. Finally, Section V concludes the paper and presents future work.

## II. BACKGROUND AND RELATED WORK

### A. Monitoring platforms for 5G networks

Standardized design and development models for 5G and B5G networks monitoring frameworks do not exist at present. Nevertheless, these topics are the subject of intense research and some proposals have already been put forward. The research is driven by the B5G systems evolution, that is characterized by increased variability including Edge segments. The proposals include the definition of a publish-subscribe mechanism to distribute data between different entities in Edge-based deployments, as in [19] and [20]. Different organizations, such as 3GPP [21], O-RAN alliance [22] and ETSI [23], integrated monitoring and data collection functions in their infrastructure models to enhance control, management, and orchestration of mobile networks, particularly regarding 5G environments and their evolution towards 6G.

In [6], [15], the design principles of the monitoring architecture under study are presented, together with a preliminary performance evaluation to characterize the platform in terms of some performance parameters, such as latency and packet loss. In [13] the platform, shown in Figure 1, is used to carry out a preliminary performance comparison between different virtualization technologies, including serverless-related technologies, such as Kata Containers or Firecracker, but not including other popular serverless platforms as OpenFaaS, which will be considered in this work to redesign some modules of the target platform using a serverless approach, feasible for Beyond 5G systems. More in general, the purpose of the monitoring platforms proposed in these papers is essentially to monitor all infrastructure and application metrics and KPIs, with a system based on a publish-subscribe mechanism to collect and distribute monitoring data through the platform. In this way, as long as the components can provide the monitoring data to this system, these metrics and their corresponding KPIs can be monitored consequently.

The original platform, as well as its evolution towards serverless computing presented in this work, takes into account both multi-site and multi-stakeholder scenarios. The experiments run by different stakeholders are differentiated through the generation of different topics, and monitoring data are published to them. An experiment, including not

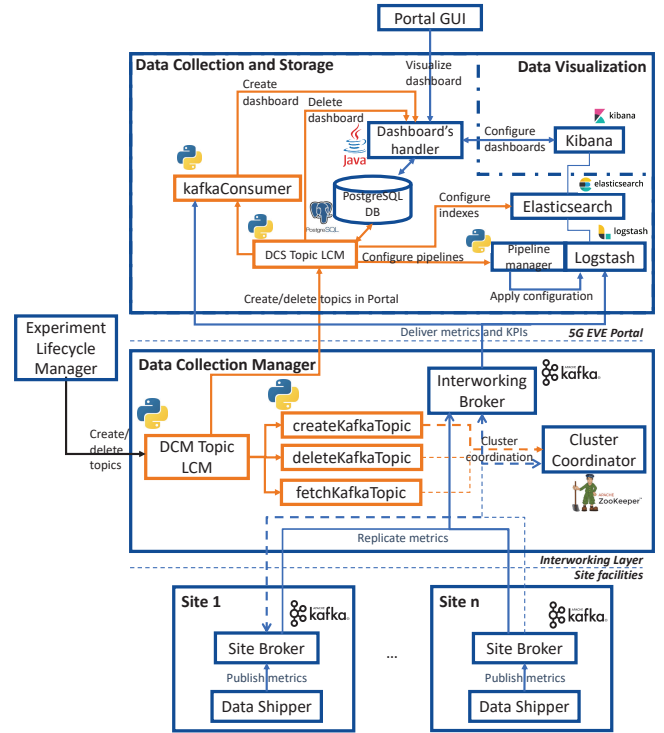


Fig. 1: Monitoring architecture designed with microservices [13]. Functions candidate for serverless implementation are highlighted in orange.

only the service to test, but also additional components emulating different context conditions, such as traffic load and delay, runs in a specific 5G slice, which identifies a portion of communications and computing resources dedicated to a stakeholder. Different experiments can run in the same slice or in different slices, depending on their nature. For example, a mobile broadband experiment will likely run in a slice different from that designed for running massive IoT services. However, this is oblivious to the monitoring platform. In fact, the topics relevant to an experiment are defined and created when it is instantiated. This enables the separation of different experiments at the monitoring level, by keeping the monitoring data of different experiments separated. The flexibility of the pub/sub mechanisms also offers the possibility to easily group data from different experiments together, to have either a global view or a view relevant to a given set of experiments, such as those running in the same slice.

Thus, the analysis presented in this paper stems from the work carried out in [6], [13], [15]. We will expand the analysis by addressing both data plane and management plane for the monitoring platform, evaluating the most suitable technology choices for each environment.

### B. Microservices and serverless technologies

Microservices represent an architectural style for distributed systems, in which large and complex applications are split into one or more interacting services, usually built on top of Platform-as-a-Service (PaaS) environment. Microservices

can be deployed independently and are loosely coupled. In this way, they enable frequent and rapid deployments. Each microservice may focus on a single, well defined task, representing a small business function. They can be implemented in any programming language. As such, it is possible to select the one that best fits the function to be realized. Instead, they need standardized language-neutral application programming interfaces (APIs) to communicate, such as Representational State Transfer (REST). Microservices are typically deployed in containers, such as Docker [18] containers, and orchestrated via container management platforms, among which Kubernetes [17] is a popular choice. Kubernetes handles so-called pods, which are collections of one or more containers grouped together to increase resource sharing efficiency. A pod is the smallest unit of a Kubernetes application. Indeed, Kubernetes containers in the same pod share the same compute resources, which, in turn, are pooled together to form clusters. These clusters can provide a more powerful distributed system for executing applications. The mapping of each service of a microservice architecture into containers and pods is left to the developer, considering resource usage and service interactions. Additional details can be found in [24].

Serverless computing is an emerging paradigm wherein computing resources are dynamically provisioned at run-time, through an event-driven allocation control, typically referred to as *Function as a Service* (FaaS). The FaaS model requires application logic to be implemented as atomic functions, which may be written in various languages and packed with their modules and dependencies. These functions are then exposed as web services, or triggered by specific events or conditions. Upon invoking a function, the platform runs it through a container, either reused from previous executions (warm start mode) or a new one (cold start mode) [11].

Compared with a conventional Infrastructure-as-a-Service deployment (IaaS), the FaaS model ensures need-based resource allocation, deploying containers on demand and possibly in parallel. It does so without any need of controlling application deployment processes at the operating-system level. This means that containers are dynamically scheduled in the hardware or in the virtual infrastructure maintained by cloud providers. Note that FaaS is intended to be different from *Software as a Service* (SaaS), due to its event-based architecture and auto-scalability, emphasizing its virtual independence from the underlying servers (*i.e.*, serverless). This contrasts with traditional methods in which the needed resources are planned during application design and provided during deployment. The name serverless computing highlights that application design does not include any production servers but can focus solely on the application code, organized through individual functions executed in microVMs or containers. Serverless computing was identified as a promising approach for several applications, such as those for data analytics at the network Edge [25] [26].

However, these serverless frameworks and techniques are somehow tied to the platform in which they are implemented, as providers intend to lock-in their serverless clients by also offering extra services that assorts the provisioning of serverless applications [27]. For this reason, serverless frame-

works are becoming increasingly popular, having the purpose of hiding the technical features of the serverless platform or cloud infrastructure for application developers. This abstraction simplifies the process of designing, developing and deploying the serverless functions [28]. Examples of these frameworks, in terms of open-source solutions, are *OpenFaaS*, *Kubeless*, *Fission* or *OpenWhisk*, among others. OpenFaaS [16], the selected platform for our serverless testbed, is a cloud native computing foundation (CNCF) open source serverless framework. It allows developers to define and use templates of different languages to create and build serverless functions. It relies on Docker images and Kubernetes control plane to run applications, providing fail-over, high availability, scale-out and security management. In the OpenFaaS framework, the OpenFaaS gateway, which is similar to a reverse proxy, is in charge of exposing and managing the function pods, offering a REST API for all interactions.

The decision of resorting to any specific solution essentially depends on the vertical stakeholders expected to be hosted on the system to be developed, as each framework offers different capabilities that make it suitable for particular use cases. The paper [29] identifies some key aspects of OpenFaaS, highlighting its easy extensibility and good performance throughout different scenarios. It shows the best flexibility in supporting multiple container orchestrators and largest adoption rate among the analyzed serverless frameworks. The work presented in [30] shows a performance analysis of a serverless IoT application deployed by OpenFaaS, considering warm and cold start approaches. This analysis shows the advantages and disadvantages of both deployment modes, highlighting the serverless flexibility and low resource requirements. However, a comparison with other paradigms, such as the mere use of microservices, is not provided.

In this paper we analyse the performance of the popular open-source serverless framework, OpenFaaS, for a realistic deployment, consisting of a 5G monitoring platform. We present the comparison between serverless and microservice-based deployments, and analyze the impact of cold and warm start modes.

### III. ARCHITECTURE OF THE 5G MONITORING PLATFORM

To fully describe the implications of adopting the serverless paradigm, the 5G EVE monitoring platform is used as an example of a 5G-related system that is liable to be transformed into serverless in some of its main building blocks. The original system includes multiple 5G network sites belonging to different operators. Nevertheless, it could be used for representing more general deployments, in which a single operator deploys a pub/sub infrastructure in each Edge node. In the current deployment, all data coming from the Edges are aggregated into a central cloud broker, connected with the system in charge of consuming the data and generating metrics and KPIs per experiment.

This platform was designed as a modular architecture, as depicted in Figure 1. Two main building blocks were defined: the Data Collection Manager (DCM), which is in charge of managing both the publish-subscribe delivery system and the

monitoring data collection, and the Data Collection Storage and Data Visualization (DCS-DV), providing indexing and visualization mechanisms.

The complete *Broker System* of the considered platform is divided into a set of *Site Broker* entities (one for each site), playing the role of the *Intra-site broker system*, together with the *Interworking Broker* entity in the *Data Collection Manager*, which represents the *Inter-site broker system*. All the brokers are based on *Apache Kafka*, and are coordinated and orchestrated by the *Cluster Coordinator*, which is based on *Apache ZooKeeper*<sup>2</sup>.

The main component that implements the operations needed to publish the metric data is the Data Shipper. It is logically connected with the Multi-Broker cluster, that is used for executing the log-to-metric operations. These operations convert metrics with heterogeneous formats, even raw logs, obtained from components and collection tools, into metrics with a uniform format.

The Broker system is mapped into a set of publish-subscribe queues, starting from the local queues deployed in the *Intra-site broker*. The queues aggregate metrics into the Interworking publish-subscribe queue (*Inter-site broker*). This broker provides to components in upper layers a transparent access to data from all sites. Each *Intra-site broker* is represented by a *Site Broker* entity, located in each site, that forwards the received data to the *Data Collection Manager*. The latter is in charge of providing the interested entities with the requested data coming from different sources.

In the DCM, four functions are considered as candidate serverless modules. These functions, modeled as REST-based services, are depicted in Fig. 1 (central block in the figure):

***createKafkaTopic***, which is in charge of creating topics in *Kafka*.

***deleteKafkaTopic***, which is in charge of deleting topics in *Kafka*.

***fetchKafkaTopic***, an auxiliary module that allows checking whether a given topic already exists in *Kafka*.

***DCM Topic LCM***, also referred to as ***dcm***, which manages the other three functions and the calls to the equivalent module in the Data Collection and Storage-Data Visualization component, used to orchestrate the configuration of the whole platform. This component has two related operations : (***dcmCreate***) and (***dcmDelete***), depending on whether the configuration has to be created or deleted, respectively.

The DCS component, which collects each of the subscribed components data through *Logstash*<sup>3</sup>, provides data persistence, searching and filtering capabilities. It does so for obtaining the useful data to be monitored during the operation of the system thanks to *Elasticsearch*<sup>4</sup>. For this component (block at the top in Figure 1), two REST-based functions can be implemented as serverless modules:

***kafkaConsumer***, which creates a *Python*-based *Kafka* consumer listening to the topic created in the platform,

with the function of triggering the creation of the corresponding Kibana dashboard when the first message is received in the topic. This way the dashboard is created only when data are available in the topic.

***DCS Topic LCM***, which is in charge of configuring the modules of DCS, based on the Elastic Stack, when the *DCM Topic LCM* sends the proper request to this component, which can trigger two different operations: (***dcsCreate***) and (***dcsDelete***), depending again on whether the configuration has to be created or deleted, respectively.

The *Data Visualization* component allows monitoring the progress of the deployment. Monitoring data can be displayed through *Kibana*<sup>5</sup>, by using a set of dashboards that are created for each deployment. A complete description of the modules composing the platform is included in [6].

The monitoring platform was originally designed by a microservice pattern to provide all the required functions by using Docker containers orchestrated by a lightweight Kubernetes version [6]. However, the original design considers two levels of virtualization, since the platform is hosted in a VM, where containers are deployed. Thus, resource management and service capability are impacted by the presence of VM virtualization. Our goal is twofold: first we identify the components that, being event based, can be transformed into serverless functions. Secondly, for components not suitable for being deployed by using the serverless model, we evaluate alternative virtualization technologies to improve resource usage and service latency.

#### A. Identification of Candidate Serverless Functions

As for the first goal, we identified a set of handlers defined in both components DCM and DCS, previously introduced and framed in orange in Figure 1. These handlers are in charge of managing the lifecycle of the topics related to the metrics and KPIs for any experiment. They do so by triggering actions such as creation of a topic in *Apache Kafka*, or building of a Kibana dashboard for a given topic. They result in atomic functions that interact through a sequence of operations used to obtain the desired result. Each handler is implemented as a REST HTTP service.

The handlers described above are feasible candidates to be re-designed as serverless functions due to the following features:

Event oriented: The handlers perform their function only when a new experiment is created or deleted in the platform

Stateless nature: Multiple executions of the same handler are independent of each other, a common state can be kept in a remote storage system (*e.g.*, *Kafka* for storing the topics, *PostgreSQL* for storing created dashboards).

Removability in idle mode if not used: Since the handlers are not needed for the whole duration of the experiment, their execution environment can be deallocated to save infrastructure resources.

<sup>2</sup><https://zookeeper.apache.org/>

<sup>3</sup><https://www.elastic.co/logstash/>

<sup>4</sup><https://www.elastic.co/elasticsearch/>

<sup>5</sup><https://www.elastic.co/kibana/>

According to the discussion on the atomic functions described above and depicted in Fig.1, two different service function chains can be identified as candidate to be made deployable through a serverless approach:

**Creation chain:** It is triggered by the *dcmCreate* operation. This operation makes use of the *fetchKafkaTopic* function to check whether the *Kafka* topic to be created exists. If not, it is created by the *createKafkaTopic* function. Then, a notification to the *dcsCreate* function is triggered to configure the Elastic Stack and to create a *kafkaConsumer* in an asynchronous way.

**Deletion chain:** It is triggered by the *dcmDelete* operation. This operation also makes use of the *fetchKafkaTopic* function to check whether the *Kafka* topic to be created exists. In this case, only if it exists, it is deleted by the *deleteKafkaTopic* function, and then a notification to the *dcsDelete* function is triggered. This is to remove the configuration from the Elastic Stack.

In the next section we describe the process of transforming the candidate microservices into serverless functions while trying to meet the challenges of the serverless paradigm proposed in [31] by using the OpenFaaS platform.

All the functions identified above are related to management tasks of the monitoring platform, and more specifically to its configuration. In more detail, they are event-triggered functions, invoked when a new experiment is created, and once again when it is deleted. Hence, they are the ideal candidates for serverless implementation, to save computing resources when they are idle. As for the delivery of the data monitoring messages, the main component involved in data transfer is the broker. This is not a good candidate for serverless deployment, since the serverless paradigm uses an application gateway, that has to invoke the appropriate functions to be executed each time a new request arrives. In the case of the broker, each new message containing monitoring data should pass through the gateway, which could become the system's bottleneck. In fact, the rate associated with the stream of the monitoring data is in the order of 100 Mbps. For this component, we evaluate alternative virtualization technologies, ranging from bare metal (no virtualization at all, Kafka broker installed on the hardware), to VM-based virtualization through Kernel-based Virtual Machine (KVM) [32], to different container technologies, to improve service capacity, service latency, and scalability. This analysis is reported in Section IV-B. In this context, it must be noted that it is possible to do a composition of both microservices and FaaS, communicating with each other through REST API calls. This strategy is recommended when modules have different latency requirements or when the system includes both event-triggered and always-on services. In fact, the final configuration of the considered 5G monitoring platform includes the cooperation of microservices, which handle data plane of monitoring services, with FaaS modules, which leverage serverless paradigm's benefits to deploy the handlers meeting the aforementioned criteria.

### B. Transformation into Serverless Architecture

The REST-based services identified in the previous section are modeled as serverless functions managed by *OpenFaaS*.

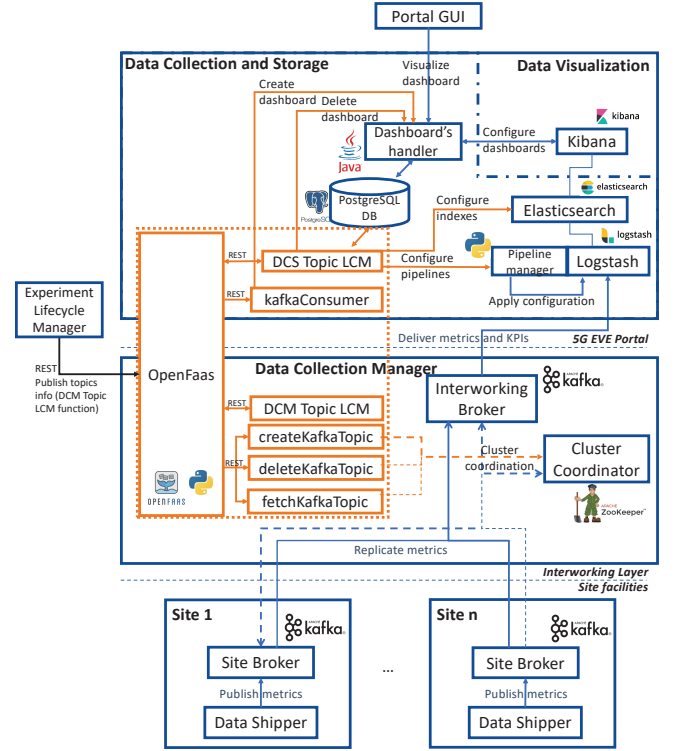


Fig. 2: Introduction of serverless functions (framed in orange) in the target monitoring platform.

This transformation can be observed in Figure 2, in which the serverless components are framed in orange. Consequently, the workflows change considerably, as the *OpenFaaS* platform becomes the central component of the serverless implementation. *OpenFaaS* exposes the interfaces needed to reach each serverless function. When an entity requests a service offered by a serverless function, it contacts the *OpenFaaS* gateway. That gateway will then trigger the instantiation of the function (cold start) and route the request to the relevant container. *OpenFaaS* is in charge of managing the lifecycle of the serverless function, thus instantiating resources and releasing them when the execution is completed. The shift from microservices to FaaS modules leads to significant savings in code and application logic development time. As presented in Cloud Native Computing Foundation's serverless whitepaper [12], serverless enables developers to focus on the logic of atomic modules and let the platform take care of the rest (i.e. trigger-to-function logic, information passing from one function to another function, auto-provisioning of container, autoscaling, identity management, etc.). This reduction of responsibilities provides lower requirements for infrastructure management compared to the ones of any of the cloud native paradigms. IaaS users still have to manage their virtual resources or, when dealing with PaaS providers, to configure the application to match the PaaS requirements [33]. Serverless applications developers in contrast do not have to consider the operating or file system, runtime execution management, and even container management.

Code 1 in the appendix shows the implementation of the *cre-*

*ateKafkaTopic* module as a Python microservice. The reader can notice that several lines of code are not directly related to the business logic of the service but to the application setup, configuration, and execution. Lines 1-9 are used to import packages for the microservice and many of them are not strictly needed for the implementation of the business logic. For example, *flask* is a web framework needed to declare HTTP paths, *argparse* manages command line arguments and *logging* and *colored* support the logging of the application. The functions covered by these packages are quite common in any microservice built on an HTTP API. Lines 12 and 13 initialize the *flask* application and logging. Lines 16 to 21 defines an endpoint to provide users with some documentation about the microservice. Lines 24 to 50 contain the implementation of the endpoint of the microservice, implementing the *createKafkaTopic* function. Finally, from line 53 to 85, the initialization of the application happens. This includes: (i) parsing and validating the command line arguments, (ii) setting up the logging, and (iii) serving the application and configure its IP address and port. The part strictly related to the business logic of the microservice i.e., the creation of topics (*create\_kafka\_topic()* function) represents less than one third of the overall service logic, 25 lines of code out of 85.

Code 2 in the appendix shows the same *createKafkaTopic* function developed in a serverless way using OpenFaaS' *python3-http* template<sup>6</sup>. The reduction in terms of lines of code with respect to the microservice version is clear: Exploiting OpenFaaS of-watchdog<sup>7</sup> and the transparent function's endpoints lifecycle management, the only logic needed for the *createKafkaTopic* module is the actual function with no additional overhead. Thus, the former service consisting of 85 lines of code is reduced to 32 lines. The *handle()* function reproduces the same logic as the *create\_kafka\_topic()* function in Code 1 (some refactoring has been applied), while the other functionalities (e.g., web framework, logging, etc.) are provided by the OpenFaaS platform. Moreover, the translation of a microservice into a FaaS serverless function provides an additional benefit to the developers: Horizontal scalability is completely handled by the serverless platform, with no need to manage the process at code level.

To sum up, the evolution from microservice to serverless allows saving of at least 62% in lines of code, which reflects not only in a significantly reduced software implementation effort [34], but also in code simplification and maintenance. Besides, these considerations do not include the additional code and configuration files needed to properly deploy the microservice in a production environment (e.g., web server, reverse proxy, SSL setup), which require a significant additional effort, not required by FaaS.

#### IV. PERFORMANCE EVALUATION

##### A. Testbed setup

For the evaluation process, two physical servers based on *Ubuntu Server 20.04* were used. Their hardware specifications are reported in Table I. They are connected to the same LAN

<sup>6</sup><https://github.com/openfaas/python-flask-template>

<sup>7</sup><https://github.com/openfaas/of-watchdog>

TABLE I: Specification of the servers used in the testbed.

| Server name | Server #1                                  | Server #2                                  |
|-------------|--|--|
| CPU         | Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40 GHz | Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30 GHz |
| RAM         | 128 GB @ 2133 MT/s                         | 64 GB @ 2133 MT/s                          |
| Disk        | 280 GB (145 MB/s write speed)              | 280 GB (145 MB/s write speed)              |
| Net. ifaces | 1x10 Gbps, 4x1 Gbps                        | 1x10 Gbps, 4x1 Gbps                        |

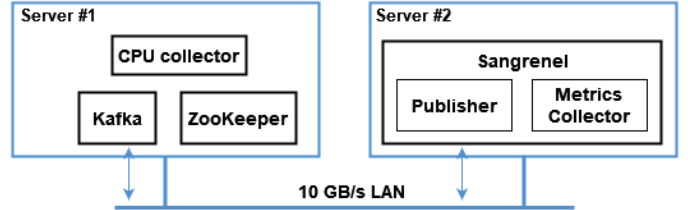


Fig. 3: Deployment of testbed components on the servers.

as illustrated in Fig. 3. Time synchronization is achieved by using the Network Time Protocol (NTP). The two servers were used for different purposes. The first one, **Server #1**, hosts the monitoring platform, and it is in charge of collecting its CPU utilization for different workloads. Instead, **Server #2** is used to generate the workload and to collect service related metrics. Specifically, in Server #1, the *Data Collection Manager* component from the Monitoring platform<sup>8</sup> was deployed for each virtualization technique studied. In particular, the only subcomponents of the DCM used for the tests are *Kafka* and *ZooKeeper*. In the tests with 2 instances of the *Kafka* broker, both of them run on Server #1, managed by *ZooKeeper*. A CPU collector script based on the *mpstat* command was used for measuring the relevant CPU consumption on Server #1. Regarding Server #2, a *Kafka* publisher based on *Sangrenel*<sup>9</sup> is used for obtaining the performance parameters under study.

The first set of tests is relevant to the evaluation of alternative virtualization components for running the *Kafka* broker of the DCM. Thus, the testbeds deployed in Server #1 to evaluate the technologies under study are the following:

**Bare-metal testbed**, using directly the Server #1 hardware without any virtualization technique. In this case, *Kafka* and *ZooKeeper* are directly installed as Linux services.

**KVM testbed**, deploying *Kafka* and *ZooKeeper* in a specific Ubuntu virtual machine<sup>10</sup>. This is the default setup, analyzed in detail in [6].

**Docker testbed**, using *runc* as container runtime (the default one for Docker) for deploying *Kafka* and *ZooKeeper*.

**Kubernetes testbed**, using *Kubernetes* (v1.19)<sup>11</sup>, with *containerd* as container runtime, in order to deploy pods running *Kafka* and *ZooKeeper*.

<sup>8</sup>The software related to this architecture is publicly available in the 5G EVE Github repository: <https://github.com/5GEVE>

<sup>9</sup><https://github.com/jamiealquiza/sangrenel>

<sup>10</sup>The disk used for the VM, based on the qcow2 technology, is configured with the writeback mode for the cache, and also using the metadata property for the preallocation parameter.

<sup>11</sup>The *Kubernetes* distribution does not affect the performance of the container runtime.

The results of the evaluations are reported and analyzed in Section IV-B.

For the investigation of the suitability of the serverless paradigm to implement the management part of the 5G EVE Monitoring platform<sup>12</sup>, management modules have been deployed by using Docker containers, all orchestrated by Kubernetes. The whole platform has been deployed on Server #1. In more detail, the testbeds deployed to evaluate the technologies under study are the following:

**Microservices testbed:** using *Kubernetes* (v1.19), with *Docker* as runtime, to deploy all the containers that compose the Monitoring platform.

**Serverless Functions testbed:** using *OpenFaaS* on top of *Kubernetes* (v1.19) for managing the six atomic functions described in Section III-A, exploiting *faas-netes* for this purpose. *faas-netes* is an OpenFaaS provider which enables Kubernetes for OpenFaaS.

The results of the latter set of evaluations are illustrated and analyzed in section IV-C.

### B. Experiments on virtualization technologies to handle monitoring data

1) *Evaluation plan:* The evaluation process involves a maximum number of six experiments running simultaneously on the monitoring platform.

Each experiment generates monitoring data related to both the infrastructure and the user application. Infrastructure related metrics include CPU usage, memory usage, disk usage, and network usage measured for each Virtual Network Function (VNF) composing the service. Application related metrics report information specific to the user application. They are defined by the vertical user and can include, for example, the number of devices currently connected to the service, the number of requests served per second, the number of queries to a database, round trip delay, re-transmitted packets and so on (more information on the definition of application related metrics is available in [34]).

The evaluation of each testbed mentioned above is done by emulating the registration of experiments on the monitoring platform and by generating synthetic monitoring data for each of them. Metrics are sent in the form of messages to a single instance of *Apache Kafka* after being generated with *Sangrenel*. Each experiment has its own set of topics. The generation of messages was modeled from empirical observations carried out in the 5G-EVE project and pushed toward worst case scenarios, in order to stress the monitoring platform as much as possible during evaluation. Specifically, each experiment produces 20 metrics, each one mapped into a topic of Kafka messages. The workload is composed as follows:

8 of these topics have packet size of 100B and a rate of 1000 messages per second.

8 of these topics have packet size of 1000B and a rate of 1000 messages per second.

2 of these topics have packet size of 100KB and a rate of 1 message per second.

2 of these topics have packet size of 1MB and a rate of 1 message per second.

Thus, data for simple metrics (e.g., CPU usage) was modeled with 16 topics with messages of small packet size generated at high rate, providing a view of the system status with 1 ms granularity. The remaining 4 topics model more complex data possibly coming from application related metrics, like application logs, such as the 2xx, 3xx, and 4xx response messages from selected HTTP service endpoints. Thus, the period is increased to 1 s. The resulting traffic rate generated by each experiment through its 20 topics is approximately 102.4 Mbps. These values of the period are much lower than those typically used for monitoring functions in cloud providers such as Azure [35] and Amazon [36], which use monitoring periods in the order of 1-5 minutes. This is due to two reasons. The first one is that a 5G system is much more complex than a cloud, and thus it may need a tighter monitoring. The second is that, as mentioned above, we need to stress our system, thus we artificially reduced the period of transmission of monitoring data. This implies that each of the experiment used in our evaluation can represent tens of real experiments. However, this does not endanger in any way the obtained results.

In the following, we summarize the parameters that fully define the workload used for evaluation.

**Design parameters:** They are related to input data to the system in order to properly configure the monitoring platform for the evaluations. We can distinguish between:

- **Fixed:** Each experiment publishes on 20 topics, with a concurrent publication rate of approximately 102.4 Mbps per experiment. The test duration is 5 minutes, and the number of test repetitions is fixed to 10. We show that this experiment duration is enough to obtain stationary values in Section IV-B2.
- **Variable:** There are 20 topics per experiment. The number of experiments is varying from 1 to 6. This parameter determines the aggregate throughput received by the monitoring platform, thus varying from 102.4 to 614.4 Mbps.

**Performance metrics:** these are the parameters measured during the execution of the tests, which can be:

- **CPU consumption [%]:** measured on Server #1 with the CPU collector. For having similar results on all testbeds, all the tools that are not going to be used for a particular testbed have been turned off.
- **Batch write latency [ms]:** the time spent until an ACK message is received from the *Kafka* broker, which indicates that the message has been correctly handled by the broker. This is measured on Server #2.
- **I/O message ratio:** the received throughput divided by the publication rate (ingress load), which is a measure of the fraction of messages lost by the broker due to overload. This is measured on Server #2 as well.

<sup>12</sup>The software related to this architecture is publicly available in the 5G EVE Github repository: [https://github.com/5GEVE/monitoring\\_dockerized\\_environment](https://github.com/5GEVE/monitoring_dockerized_environment)

2) *Results*: Figure 4 shows the results of the first part of the performance evaluation. It shows performance metrics (defined in Section IV-B1) for different virtualisation techniques. Four virtualization schemes for the broker are compared: bare metal, KVM-based VM virtualization, Docker containers, and Kubernetes with containerd runtime. The comparison is plotted as a function of the number of concurrent experiments in terms of a) CPU consumption (%), b) batch write latency (ms) in log scale, and c) normalized throughput. The abscissa reports the number of concurrent experiment (representative of the offered load). All subplots include the 95% confidence intervals, although they are often very small and difficult to distinguish.

Figure 4.a reports the evolution of CPU consumption versus offered load. The first observation is that a saturation effect can be noticed, a phenomenon previously observed in [15]. This means that the CPU consumption increases with an increase in the number of experiments deployed until a hard limit is reached (*i.e.*, the saturation point), which is around 25-27% for the *bare-metal*, *KVM*, *Docker* and *Kubernetes* testbeds. Comparing all the deployment alternatives, the following trends are observed:

The *bare-metal*, *Docker* and *Kubernetes* testbeds present a similar trend<sup>13</sup>, starting with a CPU consumption of 8-10% for one experiment and reaching the hard limit between 3 and 4 experiments deployed in the system.

The *KVM* scenario has a higher consumption profile at the beginning, as it saturates sooner (with 2 experiments), but it eventually reaches the same values as the previous case.

To sum up, the best performance is provided by *Kubernetes* technology, which saturates at 5 experiments (about 500 Mbps of offered load) with a CPU consumption of 25%.

The latency of the batch *FA0C4* operation is depicted in Figure 4.b. We used the log scale in order to distinguish the four curves. The latency increases with an increase of the number of experiments deployed. For this parameter, three different trends are observed, but with different implications compared to the CPU consumption:

Once again, the best results are observed for the *bare-metal*, *Docker* and *Kubernetes* testbeds, with a batch write latency value lower than  $3 < B$  in the worst case (*i.e.*, with 6 experiments deployed).

When using *KVM*, we yield latencies of one order of magnitude larger than the previous case (around  $40 < B$  in the worst case). This happens since the hypervisor's access to disk, required by Kafka, differs from the one with containers. Containers share resources with the host server. Hypervisor's access to disk obviously also differs from direct access to the disk, which is the case of the *bare-metal* scenario. Again, saturation values for this metric are reached for 2 experiments in the VM-based virtualization.

<sup>13</sup>In the case of the analysis from [15], where containers are deployed within VMs, the results are similar to the *KVM* testbed from this analysis, as containers adapt their performance to the environment in which they run.

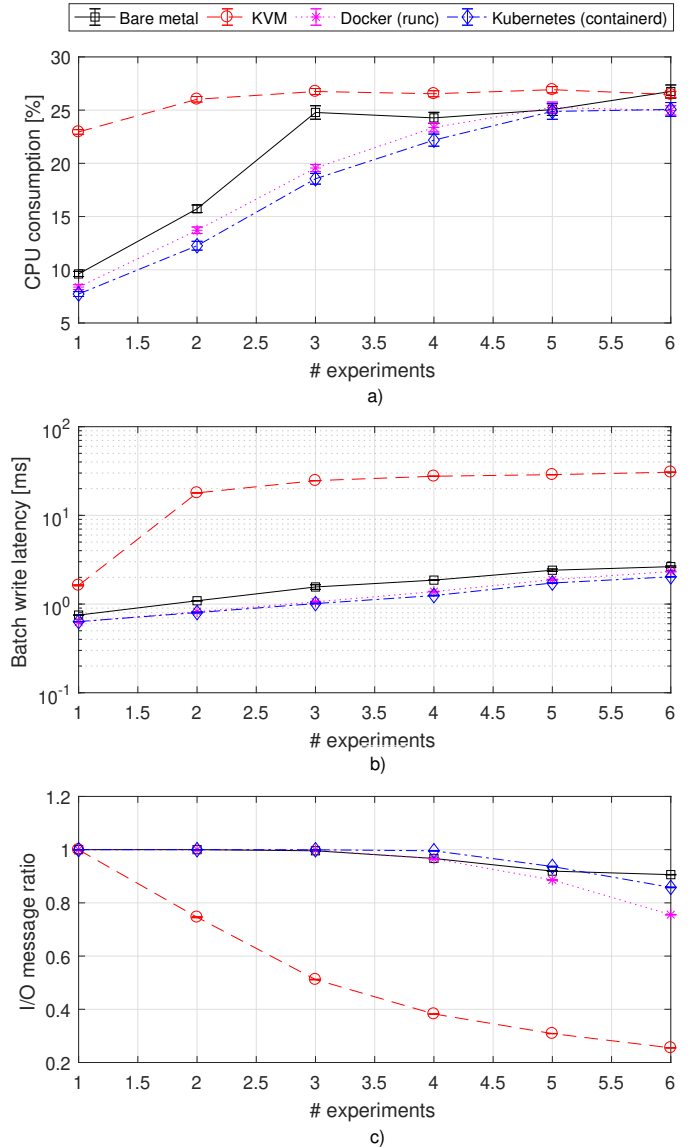


Fig. 4: Comparison between bare metal, KVM-based VM virtualization, Docker, and Kubernetes: a) CPU consumption, b) batch write latency, and c) normalized throughput.

To sum up, also in this case the best performance is reached by the *Kubernetes* technology, although it is very close to the one relevant to *Docker* and *bare-metal*.

Finally, according to the results shown in Figure 4.c, the evolution of the normalized throughput (*i.e.*, the handled throughput divided by the publication rate) depends also on the saturation effect experienced in the *Kafka* broker. When it appears, packet losses occur, causing a reduced I/O message ratio with increased number of experiments (*i.e.*, the throughput received by the platform). In this way, the three trends observed for the CPU consumption are also repeated in this case with a clear correlation between results:

First of all, the *bare-metal*, *Docker* and *Kubernetes* testbeds present the same trends and the highest possible values, with an I/O message ratio of about 0.75 in the worst case (*Docker* testbed with 6 experiments deployed).

According to the point where the I/O message rate starts falling, it is confirmed that the saturation point seems to be in the range of 3 (for *bare-metal* and *Docker*) to 4 (for *Kubernetes*) experiments deployed.

The *KVM* testbed follows, in which the saturation point is reached already with 1 experiment, as commented in the CPU consumption analysis, with a significant throughput decrease for 2 experiments, and achieving a minimum value of about 0.25 in the worst case. This trend is, in fact, the one observed in the analysis done in [15], as containers are used in a VM as host.

Also when considering I/O ratio, the best performing solution is *Kubernetes*, since it can keep nearly the 100% of throughput (*i.e.*, 1 I/O ratio) for 4 experiments (about 410 Mbps of offered load). Instead, for 4 concurrent experiments this ratio starts falling below 0.99 for *bare-metal* and *Docker*. In more detail, it falls below 0.97, which makes the configuration no longer acceptable, since a significant fraction of monitoring data gets lost. Data loss must be avoided since these data are used by the verticals to evaluate their KPIs. Thus, looking back to results in Fig. 4.a and 4.b, one can see that *Kubernetes* is the best solution, since it reaches the saturation point later and it can offer the best performance in latency and CPU consumption compared to the other solutions. As for the VM-based virtualization, it is now clear that saturation is reached in the range of 1 (0.999 I/O ratio) to 2 experiments (0.746 I/O ratio). This implies that its performance in term of CPU consumption and latency does not significantly change with more than 2 experiments.

Given these results, we decided to analyze also the scalability of the compared solutions when the same hardware is used. It is known that deploying multiple instances of the same virtualized software on the same hardware is often convenient also in terms of performance (e.g. see [37]), allowing to better exploit the computing capacity. Clearly, for the *bare-metal* solution this is not possible. As for the other solutions, we considered *KVM* virtualization and, for the container solution, *Kubernetes*, given its superior performance with respect to *Docker* with a single instance. In our analysis, we considered the comparison between 1 and 2 instances for both technologies, with both instances running in Server #1. Results are shown in Figure 5, which reports a) CPU consumption in percentage, b) batch write latency in ms, and c) I/O ratio (c), always as a function of the number of experiments. Again, 95% confidence intervals are reported in all plots. Results are as expected. Both *KVM* and *Kubernetes* benefit of multiple instances, although a clear gain is obtained only with the latter. In fact, with 2 VMs it is possible to mitigate the saturation effect for 2 experiments, with an I/O ratio equal to 0.95 (see Figure 5.c), but not enough to justify its usage. Clearly, latency is improved (see Figure 5.b) due to a higher CPU consumption, which witnesses that complete saturation is still not reached. However, the highest I/O ratio is provided by *Kubernetes*. In fact, Fig.5.c shows that, when considering 2 instances of the *Kafka* broker, automatically managed by the platform, it is possible to reliably sustain the offered load of 6 experiments (0.998 I/O ratio), corresponding to about 614 Mbps. For higher

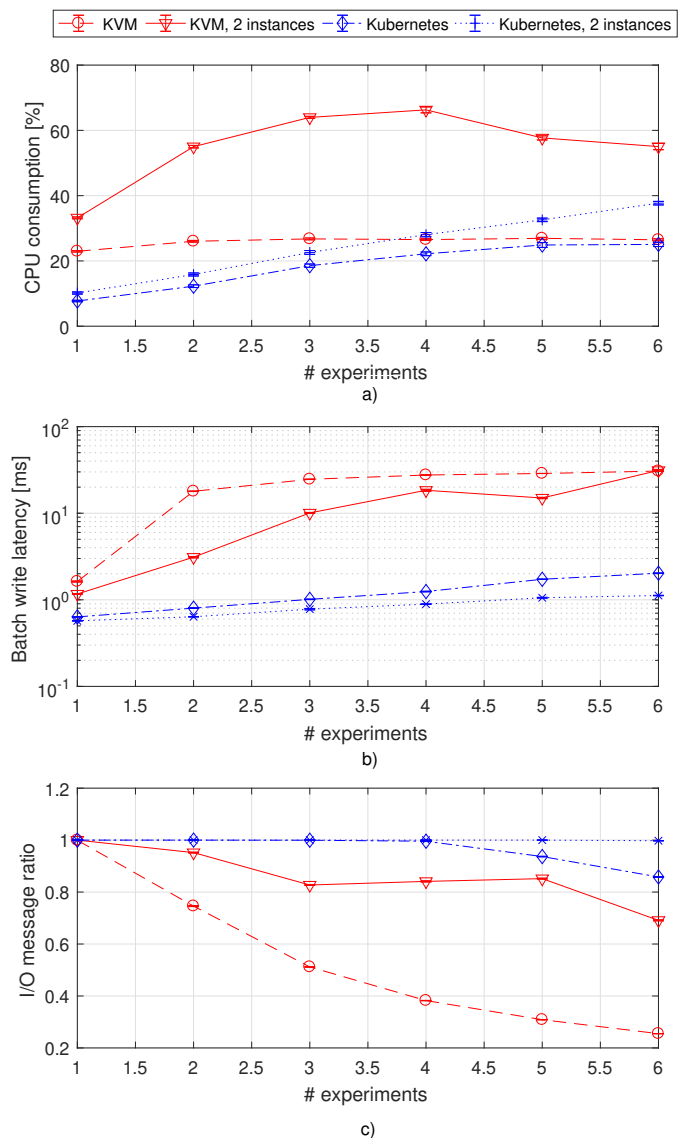


Fig. 5: Scaling comparison between *KVM* virtualization and *Kubernetes*: a) CPU consumption, b) batch write latency, and c) normalized throughput.

load, we expect more significant message loss. This is also confirmed by the CPU consumption (5.a), which increases steadily with the number of experiments, without any significant flattening. Finally, also latency benefits of this alternative deployment become visible, with a maximum latency of about 1 ms. An important remark is on the CPU consumption. Even using 2 instances, the overall CPU consumption is below 40% in non-saturated conditions (6 experiments), which is nearly the same CPU requirement as that of the *KVM* solution (2 VMs) when loaded with 1 experiment only.

It must be noted that using different topics in the same broker to separate experiments is effective only at the functional level. When the number of experiments increases, packet losses affect all experiments indistinguishably, thus not providing performance isolation for monitoring data collection. However, with the last analysis, we have identified the way to address this issue. The most straightforward solution is to

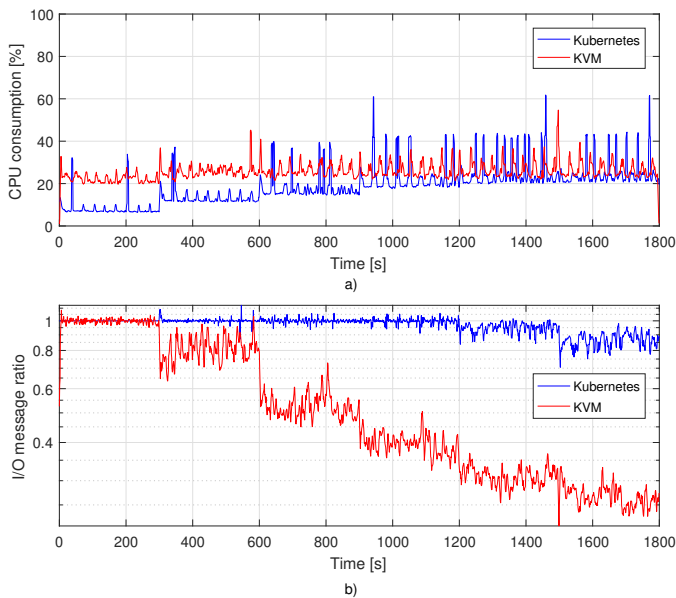


Fig. 6: Comparison of the impact of transient between KVM virtualization and Kubernetes on: a) CPU consumption, b) normalized throughput.

allow Kubernetes to scale automatically, thus replicating the broker to maintain a given performance level. In addition, since *Kubernetes* can allocate resource quotas to namespaces [38], in some scenarios it could be convenient to deploy different instances of the broker in different namespaces for handling experiments running in different slices, each one with its own computing resources. This can be done at runtime during the set up of a slice. In both ways, the data plane performance of the monitoring platform would be independent from the number of concurrent experiments, and it would be possible to exploit not only powerful servers, but also clusters of them. As for the overhead, we expect a small increase, as it appears in Figure 5.a, where for running 2 experiments (i.e. the case of interest) the CPU consumption of Kubernetes with 2 instances is only 3% higher than in the case of handling the whole traffic with a single instance.

Finally, we analyze the transient effects on CPU consumption and normalized throughput, which are depicted in Fig. 6. Figure 6 shows a) the CPU levels and b) the related I/O rate as a function of the test time for a single runtime test. During the test, the value of each workload is kept constant for 5 minutes, after which a further experiment is added, which corresponds to an increase in the workload rate of 102.4 Mb/s, up to a maximum of 6 concurrent experiments. The figure shows the performance for both Kubernetes, using the containerized runtime, and VMs managed by KVM. We can observe a step in the CPU consumption every 300 seconds, corresponding to the increase in the workload (Fig. 6.a). This phenomenon is more evident for Kubernetes, which scales much better and reaches the saturation only for 5 experiments. Considering the I/O rate, it is easy to identify the transition to higher workload levels in KVM, which enters saturation condition for 2 concurrent experiments. In addition, the transient due to a significant

increase in the workload (i.e. when a new experiment is added) always vanishes within 20 seconds, which represents less than 7% of the test duration for each workload level. This transient manifests itself with a peak of CPU occupation, which can correspond to instantaneous I/O rate values slightly lower than 100%, immediately compensated by values slightly larger than 100%<sup>14</sup>. This means that the test time is more than enough to capture the steady state of the system for each level of workload. However, from the analysis of the figure, it is possible to note that these CPU oscillations and the resulting variations in the ability to handle the offered load are present not only during the initial transient, but occur periodically throughout the test time, and the value of these spikes increases as the workload grows. These fluctuations in both CPU occupancy and I/O pace are larger with VMs than with Kubernetes containers. This is reflected in a better average performance of Kubernetes (best approach) than KVM (baseline and worst approach), as already discussed this section.

The results shown in Fig. 6 allow commenting also the dynamic behaviour of the monitored services. For instance, if the number of served users and/or their traffic increases significantly, the 5G platform scales up by adding replicas of the VNF instances, as expected. However, this implies an obvious increase in the rate of monitoring data generated by those VNFs. Also in the worst case, in which all VNFs involved in the service are replicated, this corresponds to adding a further experiment, as in Fig. 6, which can be easily managed by our platform, with a corresponding horizontal scaling operation, when needed.

### C. Experiments on management of monitoring topics

The previously described performance comparison has highlighted an overall supremacy of *Kubernetes* technology in terms of efficiency and latency. This baseline analysis has further motivated the investigation of serverless computing feasibility in a real world 5G scenario. This because the serverless computing paradigm mainly relies on containers, which are orchestrated by a system such as Kubernetes, with additional serverless-management components such as the ones in *faas-netes*<sup>15</sup>, an OpenFaaS provider which enables Kubernetes for OpenFaaS. The observed superiority of containerized approaches for the given platform motivates the choice of a microservices-designed version of the platform as a benchmark to analyze serverless computing performance in deploying the services needed by the platform. Both paradigms rely on the same virtualization technique (containerization), being different for what concerns the management and deployment of services. Microservices represent a de-facto standard architecture for containerized software deployment, whereas serverless computing is still in an early stage, making its way into application development.

<sup>14</sup>The I/O message rate can be higher than 1 because there are packets that are expected to be received at a specific time but may be delayed and received later on, so that the number of packets received at some moments can be higher than the total number of packets sent per second, but on average, the hard limit of the I/O message rate is 1.

<sup>15</sup><https://github.com/openfaas/faas-netes>

1) *Evaluation plan*: Delving into the particular tests to be executed on each testbed, they consist of the execution of the creation-and-deletion chain, already described in Section III-A, measuring the execution time spent since the request reaches the monitoring platform until it is completed. For each configuration (serverless with cold start, serverless with warm start and microservices), any execution was repeated 1.000 times, to have statistically significant results, and the execution time of each serverless function was measured by using the output of the *curl* command.

In the case of the serverless functions testbed, for both cold and warm start configurations, each execution performs a creation chain, sequentially followed by a deletion chain, while the time spent by each atomic function is measured. The execution steps are reported below. Steps 1 to 6 are related to the creation chain, whereas steps 7 to 10 are related to the deletion chain.

- 1) A topic is created with the *createKafkaTopic* function.
- 2) With *fetchKafkaTopic* function, it is checked if the previously created topic exists.
- 3) The Kafka consumer process is created with the *kafka-Consumer* function, doing an asynchronous call<sup>16</sup> to the function.
- 4) The topic is deleted with the *deleteKafkaTopic* function.
- 5) A new topic is created with another invocation of the *createKafkaTopic* function, of which the corresponding execution time is not considered.
- 6) The previous topic is used for invoking the *dcsCreate* operation from the *DCS Topic LCM*, which also implies the execution of the *kafkaConsumer* function.
- 7) The configuration created in the previous step is deleted with the *dcsDelete* operation from the *DCS Topic LCM*.
- 8) The topic is deleted with *deleteKafkaTopic*, but its execution time is not registered again.
- 9) The whole creation chain is invoked by triggering the *dcmCreate* operation in the *DCM Topic LCM*.
- 10) The whole deletion chain is invoked by triggering the *dcmDelete* operation in the *DCM Topic LCM*.

To compare the performance of the microservices testbed with that of the serverless one, and to have a clearer analysis of the whole execution chain, a test execution in the microservices testbed simply triggers the *dcmCreate* and *dcmDelete* operations.

Since the transformation of the platform into serverless involves only management components, we do not consider the batch write latency and I/O message ratio metrics for its evaluation. Having already shown the benefits in terms of development and deployment effort, we want to verify whether the serverless components can provide similar performance as their microservice counterpart. Hence, we focus on the execution time needed by a function to serve a single request and on the one needed to execute a chain of functions.

2) *Execution Time Evaluation for Serverless Functions*: The execution time measured by each serverless function in the

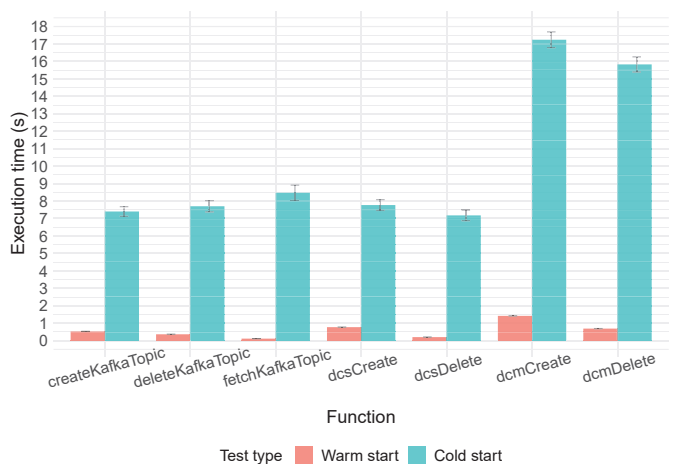


Fig. 7: Average execution time for all the serverless considered functions, in cold and warm start modes, with 95% confidence intervals.

serverless functions testbed, repeating each execution 1.000 times, is depicted in Figure 7.

From Figure 7 is clear that running functions in cold start mode leads to a significant penalty in terms of execution time. In particular, considering a simple function as *fetchKafkaTopic*, the execution time is in the order of milliseconds for the warm start mode, whereas the same function executed in a cold start mode requires more than 8 seconds to complete. Looking at the composite serverless functions *dcmCreate* and *dcmDelete* reported times, we observe that the sum of functions that belong to the same chain is higher than the execution of the whole chain, represented by the *dcmCreate/Delete* operations. For example, *dcmCreate* time should be equal to the sum of its serverless functions times (*createKafkaTopic* + *fetchKafkaTopic* + *dcsCreate*). Instead, it can be noted that the sum of simple serverless functions execution times is higher than the single execution time obtained for the composite *dcmCreate*. Hence, calling functions separately consumes more than calling functions simultaneously by invoking the corresponding composite function. This conclusion applies to both cold start and warm start modes.

To check the evolution of one serverless function for each repetition in both configurations, Figure 8 shows how the execution time of the *createKafkaTopic* function evolves for each repetition executed. Considering the graph in Figure 8, one can observe that the warm start curve remains nearly flat, whereas the cold start curve presents more variability, with spikes of up to 20 s during the considered repetitions. These spikes appear randomly, so it is an unexpected effect. This phenomenon could be related to influencing factors such as memory and CPU conditions or containers shutdown issues, as analysed in [39]. Cold start instability regarding the execution time pattern will be further investigated in future works.

3) *Execution Time Comparison between Microservices and Serverless Functions*: The comparison between the *dcmCreate* and *dcmDelete* operations in the serverless testbed (with cold and warm start) and the microservices testbed is presented in Figure 9. It shows that the obtained execution times for these

<sup>16</sup>Due to this, as the measured execution time does not reflect the real time spent by this function, it is not included in the performance evaluation.

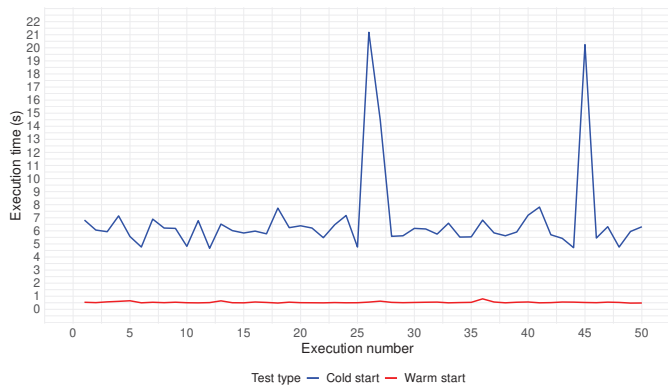


Fig. 8: Results’ comparison between the executions carried out by the *createKafkaTopic* function in cold and warm start.

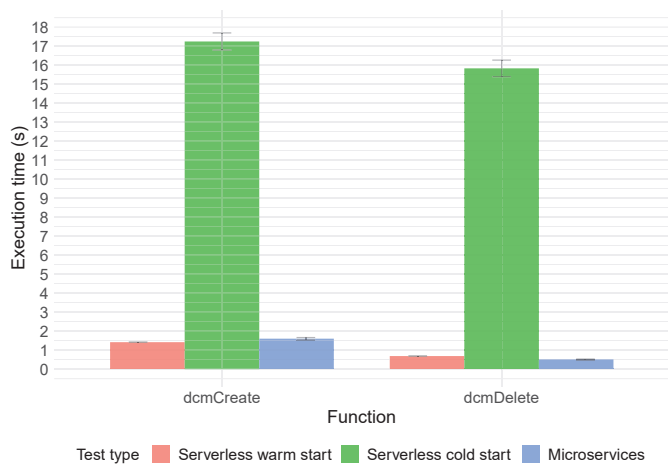


Fig. 9: Execution time for *dcmCreate* and *dcmDelete* functions in serverless and microservices deployments, with 95% confidence intervals.

two operations implemented with microservices and serverless functions in warm start mode are almost equal. Hence, a serverless approach with warm start mode can achieve the same latency performance of a microservices deployment, with the additional benefit of reduced code logic, as shown in section III-B, and infrastructure maintenance [12]. Considering a cold start approach, the same performance degradation of execution time previously detected in Figures 7 and 8 is obtained. Latency requirements for the management operations of the monitoring platform can be quite loose. Even so, several seconds of difference and, more importantly, a kind of performance instability recommend the usage of warm start mode over the cold start one. Thus, in conclusion, the serverless approach is preferable, since it allows achieving the same operation latency as a microservice implementation, but with a much easier code implementation and maintenance.

## V. CONCLUSION

In this paper, we discussed a new design of a real 5G monitoring platform by introducing serverless computing principles. We presented a performance evaluation of a FaaS

deployment and compared it to a microservices deployment in terms of execution time. Through practical examples we demonstrate the transformation of an atomic operation (i.e., *createKafkaTopic*) from microservice to FaaS modules. We highlight the benefits obtainable with the serverless paradigm. Those are essentially a simplified development workflow, significant reduction of operational costs related to infrastructure maintenance and software deployment, and almost effortless horizontal scaling. We evaluated the performance of the re-designed platform. For doing so, we have built a testbed and executed three deployment scenarios: a microservices based one, a serverless based one with warm start mode, as well as a serverless based one with cold start mode. A complete workflow of the 5G monitoring platform under study was used as test execution. We measured the execution and response time of each atomic operation. Our findings suggest that the solution of serverless deployment with warm start leads to the same performance in terms of execution time as a microservices deployment, but with the above identified additional advantages. An additional benefit of the serverless platform is improved resource usage efficiency, in particular when using a cold start approach. Nevertheless, the choice of a cold start mode entails a significant increase of execution and response times, as our analysis demonstrates. Hence, it is preferable to avoid serverless deployment in conjunction with cold start for delay-sensitive applications. Moreover, in cold start mode the overall serverless deployment suffers of some instability consisting of spikes of latency values which happened during specific tests. We plan to further investigate the nature of the cold start instabilities in future work.

However, serverless functions cannot be used to realize all components, especially those with a behavior that is not event-based. This is the case of the monitoring platform’s module in charge of handling monitoring data, that is the pub/sub broker implemented by *Apache Kafka* in our testbed. For this module, the containerized approach with *Kubernetes* using *containerd* as container runtime clearly showed superiority in terms of lower CPU consumption, lower latency, and higher throughput, with respect to the VM-based virtualization as well as to the *bare-metal* implementation. It also slightly outperformed its companion implementation using *Docker* with *runc* as container runtime. In addition, *Kubernetes* exhibits excellent horizontal scaling properties.

To sum up, the results of our analysis suggests to realize a monitoring platform for B5G systems by adopting an hybrid approach including both microservices and serverless technologies, preferring the latter when event-based operation are involved.

Future work will focus on formal modeling of the presented system, as well as on the introduction of artificial intelligence techniques to drive the horizontal scalability operations of the platform.

## APPENDIX

### CODE SNIPPETS FOR CREATEKAFKATOPIC FUNCTION

Code 1 shows the implementation of the *createKafkaTopic* module as a Python microservice, whereas Code 2 shows the

same function developed in a serverless way using OpenFaaS' *python3-http* template.

```

1 import requests
2 import argparse
3 import logging
4 import coloredlogs
5 from flask import Flask, request, jsonify
6 from flask_swagger import swagger
7 from waitress import serve
8 from kafka.admin import KafkaAdminClient, NewTopic
9 import json
10
11
12 app = Flask(__name__)
13 logger = logging.getLogger("CreateKafkaTopic")
14
15
16 @app.route("/spec", methods=['GET'])
17 def spec():
18     swag = swagger(app)
19     swag['info']['version'] = "1.0"
20     swag['info']['title'] = "CreateKafkaTopic REST
    API"
21     return jsonify(swag)
22
23
24 @app.route('/create_kafka_topic', methods=['POST'
    ])
25 def create_kafka_topic():
26     logger.info("Request received - POST /
    create_kafka_topic")
27     if not request.is_json:
28         logger.warning("Format not valid")
29         return 'Format not valid', 400
30     try:
31         admin_client = KafkaAdminClient(
32             bootstrap_servers=kafka_ip_port,
33             client_id='create_kafka_topic')
34
35
36         data = request.get_json()
37         logger.info("Data received: %s", data)
38         topic = data["topic"]
39
40         logger.info("Creating topic %s in Kafka",
    topic)
41
42         topic_list = []
43         topic_list.append(NewTopic(name=topic,
    num_partitions=1, replication_factor=1))
44         admin_client.create_topics(new_topics=
    topic_list, validate_only=False)
45         admin_client.close()
46     except Exception as e:
47         logger.error("Error while parsing request"
    )
48         logger.exception(e)
49         return str(e), 400
50     return '', 201
51
52
53 if __name__ == "__main__":
54     parser = argparse.ArgumentParser()
55     parser.add_argument(
56         "--kafka_ip_port",
57         help='Kafka IP:port',
58         default='localhost:9092')
59     parser.add_argument(
60         "--log",
61         help='Sets the Log Level output, default
    level is "info",
62         choices=[
63             "info",
64             "debug",
65             "error",
66             "warning"],

```

```

67         nargs='?',
68         default='info')
69
70     args = parser.parse_args()
71     numeric_level = getattr(logging, str(args.log)
    .upper(), None)
72     if not isinstance(numeric_level, int):
73         raise ValueError('Invalid log level: %s' %
    loglevel)
74     coloredlogs.install(
75         fmt='%(asctime)s %(levelname)s %(message)s
    ',
76         datefmt='%d/%m/%Y %H:%M:%S',
77         level=numeric_level)
78     logging.getLogger("CreateKafkaTopic").setLevel(
    numeric_level)
79     logging.getLogger("requests.packages.urllib3")
    .setLevel(logging.ERROR)
80
81     global kafka_ip_port
82     kafka_ip_port= str(args.kafka_ip_port)
83
84     logger.info("Serving CreateKafkaTopic on port
    8190")
85     serve(app, host='0.0.0.0', port=8190)

```

Code 1: *createKafkaTopic* as a microservice module

```

1 from kafka.admin import KafkaAdminClient, NewTopic
2 import json
3
4 def handle(event, context):
5     if event.method == 'POST':
6         data = json.loads(event.body)
7         if "topic" not in data:
8             return {
9                 "statusCode": 400,
10                "body": "Format not valid"
11            }
12        try:
13            topic = data["topic"]
14            admin_client = KafkaAdminClient(
15                bootstrap_servers="kafka.deployment8:9092",
16                client_id='create_kafka_topic')
17            topic_list = []
18            topic_list.append(NewTopic(name=topic,
19                num_partitions=1, replication_factor=1))
20            admin_client.create_topics(new_topics=
21                topic_list, validate_only=False)
22            admin_client.close()
23        except Exception as e:
24            return {
25                "statusCode": 400,
26                "body": ".format(e)
27            }
28        return {
29            "statusCode": 200,
30            "body": "OK"
31        }
32    else:
33        return {
34            "statusCode": 200,
35            "body": "No action for this endpoint"
36        }

```

Code 2: *createKafkaTopic* as a FaaS module

## REFERENCES

- [1] H. Xue, B. Huang, M. Qin, H. Zhou, and H. Yang, "Edge computing for internet of things: A survey," in *2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, 2020, pp. 755–760.

- [2] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, p. 27–32, Oct. 2014. [Online]. Available: <https://doi.org/10.1145/2677046.2677052>
- [3] F. Giust, G. Verin, K. Antevski, J. Chou, Y. Fang, W. Featherstone, F. Fontes, D. Frydman, A. Li, A. Manzalini, D. Purkayastha, D. Sabella, C. Wehner, K.-W. Wen, and Z. Zhou, "Mec deployments in 4g and evolution towards 5g," ETSI White Paper, Tech. Rep., February 2018.
- [4] G. Faraci, C. Grasso, and G. Schembra, "Fog in the clouds: Uavs to provide edge computing to iot devices," *ACM Trans. Internet Technol.*, vol. 20, no. 3, Aug. 2020. [Online]. Available: <https://doi.org/10.1145/3382756>
- [5] C. Casetti, C. F. Chiasserini, T. Deiß, P. A. Frangoudis, A. Ksentini, G. Landi, X. Li, N. Molner, and J. Mangues, "Network slices for vertical industries," in *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, 2018, pp. 254–259.
- [6] R. Perez, J. Garcia-Reinoso, A. Zabala, P. Serrano, and A. Banchs, "An experimental publish-subscribe monitoring assessment to beyond 5g networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2021, no. 1, Apr. 2021. [Online]. Available: <https://doi.org/10.1186/s13638-021-01962-y>
- [7] M. Plauth, L. Feinbube, and A. Polze, "A performance survey of lightweight virtualization techniques," in *Service-Oriented and Cloud Computing*, F. De Paoli, S. Schulte, and E. Broch Johnsen, Eds. Cham: Springer International Publishing, 2017, pp. 34–48.
- [8] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 1: Reality check and service design," *IEEE Software*, vol. 34, no. 1, pp. 91–98, 2017.
- [9] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, p. 44–54, Nov. 2019. [Online]. Available: <https://doi.org/10.1145/3368454>
- [10] J. B. Moreira, H. Mamede, V. Pereira, and B. Sousa, "Next generation of microservices for the 5g service-based architecture," *International Journal of Network Management*, vol. 30, no. 6, p. e2132, 2020, e2132 nem.2132. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.2132>
- [11] L. Baresi and D. Figueira Mendonça, "Towards a serverless platform for edge computing," in *2019 IEEE International Conference on Fog Computing (ICFC)*, 2019, pp. 1–10.
- [12] "CNCF WG-Serverless Whitepaper v1.0."
- [13] R. Perez, P. Benedetti, M. Pergolesi, J. Garcia-Reinoso, A. Zabala, P. Serrano, M. Femminella, G. Reali, and A. Banchs, "A performance comparison of virtualization techniques to deploy a 5G monitoring platform," in *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit): Operational & Experimental Insights (OPE) (2021 EuCNC & 6G Summit - OPE)*, Porto, Portugal, Jun. 2021.
- [14] "European 5g validation platform for extensive trials," [Accessed on 20-Sept-2021]. [Online]. Available: <https://www.5g-eve.eu/>
- [15] R. Perez *et al.*, "A Monitoring Framework for Multi-Site 5G Platforms," in *EuCNC 2020*, 2020, pp. 52–56.
- [16] "Openfaas: Open function as a service," [Accessed on 26-Dec-2020]. [Online]. Available: <https://www.openfaas.com/>
- [17] "Production-grade container orchestration," [Accessed on 20-Sept-2021]. [Online]. Available: <https://kubernetes.io/>
- [18] "Empowering app development for developers — docker," [Accessed on 20-Sept-2021]. [Online]. Available: <https://www.docker.com/>
- [19] A. Javed, K. Heljanko, A. Buda, and K. Främling, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT 2018)*. United States: IEEE, Feb. 2018, pp. 813–818.
- [20] C. Martín, D. Garrido, M. Díaz, and B. Rubio, "From the edge to the cloud: Enabling reliable iot applications," in *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2019, pp. 17–22.
- [21] "3GPP: architecture enhancements for 5G system (5GS) to support network data analytics services (release 16). ts 23.288 v16.1.0," 3GPP, Tech. Rep., 2019. [Online]. Available: <https://www.3gpp.org/DynaReport/23288.htm>
- [22] "O-RAN: towards an open and smart RAN," 2018.
- [23] "ETSI: network transformation; (orchestration, network and service management framework). White paper no. 32," 2019.
- [24] S. Daya, *Microservices from theory to practice : creating applications in IBM Bluemix using the microservices approach*. Poughkeepsie, NY: IBM Corporation, International Technical Support Organization, 2015.
- [25] S. Nastic, T. Rausch, O. Scekcic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.
- [26] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge computing: Extending serverless computing to the edge of the network," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3078468.3078497>
- [27] R. Pérez Hernández, "Monitoring and orchestration of network slices for 5g networks," Ph.D. dissertation, Universidad Carlos III de Madrid. Departamento de Ingeniería Telemática, 2021.
- [28] K. Kritikos and P. Skrzypek, "A Review of Serverless Frameworks," in *IEEE/ACM UCC Companion*, 2018, pp. 161–168.
- [29] V. Aggarwal and B. Thangaraju, "Performance Analysis of Virtualisation Technologies in NFV and Edge Deployments," in *IEEE CONECCT 2020*, 2020, pp. 1–5.
- [30] P. Benedetti, M. Femminella, G. Reali, and K. Steenhaut, "Experimental analysis of the application of serverless computing to IoT platforms," *Sensors*, vol. 21, no. 3, 2021. [Online]. Available: <https://www.mdpi.com/1424-8220/21/3/928>
- [31] M. Gramaglia, P. Serrano, A. Banchs, G. Garcia-Aviles, A. Garcia-Saavedra, and R. Perez, "The case for serverless mobile networking," in *2020 IFIP Networking Conference (Networking)*, 2020, pp. 779–784.
- [32] "Kernel virtual machine," [Accessed on 20-Sept-2021]. [Online]. Available: [https://www.linux-kvm.org/page/Main\\_Page](https://www.linux-kvm.org/page/Main_Page)
- [33] H. Shafei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges and applications," 2021.
- [34] M. Femminella, M. Pergolesi, and G. Reali, "5G experiment design through Blueprint," *Computer Networks*, p. 107948, Feb. 2021. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S1389128621000864>
- [35] "Azure monitoring," [Accessed on 26-Jan-2022]. [Online]. Available: <https://docs.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-cost>
- [36] "Amazon cloudwatch," [Accessed on 26-Jan-2022]. [Online]. Available: <https://aws.amazon.com/it/cloudwatch/pricing/>
- [37] M. Femminella, F. Giacinti, and G. Reali, "Optimal deployment of open source application servers providing multimedia services," *IEEE Network*, vol. 28, no. 5, pp. 54–63, 2014.
- [38] "Kubernetes resource quotas," [Accessed on 20-Sept-2021]. [Online]. Available: <https://kubernetes.io/docs/concepts/policy/resource-quotas/>
- [39] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," 10 2018.



**Ramon Perez** received his Ph.D. in Telematics Engineering from the University Carlos III of Madrid (UC3M) in 2021, his B.Sc. in Telecommunication Technology Engineering from the University of Seville in 2015 and his M.Sc. in Telecommunication Engineer from the Polytechnic University of Madrid in 2017. His research interests include network slicing, network virtualization, monitoring and network automation in 5G networks.



**Priscilla Benedetti** received the master degree in Computer Engineering from University of Perugia in 2019. She is a Ph. D. student of University of Perugia and of the Vrije Universiteit Brussel. Her main interest are cloud computing, virtual networks and machine learning.

