Microservices Configurations and the Impact on the Performance in Cloud Native Environments

Mohamed Mekki
Communication system department
EURECOM
Sophia Antipolis, France
Email: mohamed.mekki@eurecom.fr

Nassima Toumi
Communication system department
EURECOM
Sophia Antipolis, France
Email: nassima.toumi@eurecom.fr

Adlen Ksentini
Communication system department
EURECOM
Sophia Antipolis, France
Email: adlen.ksentini@eurecom.fr

***Abstract*—Cloud-native rethinks the application architecture by embracing a micro-service approach, where each microservice is packaged into containers to run in a centralized or an edge cloud. When deploying the container running the micro-service, the tenant has to specify the amount of CPU and memory limit to run their workload. However, it is not straightforward for a tenant to know in advance the computing amount that allows running the microservice optimally. This will impact the service performances and the infrastructure provider, particularly if the resource overprovisioning approach is used. To overcome this issue, we conduct in this paper an experimental study aiming to detect if a tenant's configuration allows running its service optimally. We run several experiments on a cloud-native platform, using different types of applications under different resource configurations. The obtained results provide insights on how to detect and correct performance degradation due to misconfiguration of the service resource.**

I. Introduction

In recent years, software development models have shifted from monolithic architectures to loosely coupled microservices. In the monolithic architecture model, all the components of the system are part of the same application making it harder to deploy, manage and improve its functionality due to the high coupling and dependence between the components of the application [1]. While in micro-service architecture, an application is decoupled into many loosely distributed services that have independent and straightforward functions following the single responsibility principle. Microservices, combined with containerization and container orchestration solutions such as Kubernetes [2], allowed the emergence of the cloud-native ecosystem. A recent model in which applications are developed to take full advantage of the distributed computing offered by the cloud.

This evolution led many vertical industry to consider migrating their applications in cloud and Edge environments [3] to get full advantage of the cloud-native deployments and the benefits that it offers. Indeed, cloud-native offers reliable and self-healing deployment as containers are deployed using advanced container orchestration solutions such as Kubernetes, Openshift, etc [4]. These solutions, in addition to the decomposability of the applications, reduce the points of failures and speed up the recovery in cases where failures occur in a set of microservices. The cloud-native has an impact not only on the vertical industry but also on other industries such as telecommunication. Indeed, the 5th generation of mobile network (5G) builds on cloud-native the 5G core network functions, known as Service Based Architecture (SBA); all the network functions are cloud-ready and can be deployed in containers on a cloud or edge infrastructure [5].

But despite this shift in application design, one fundamental problem is setting the configuration (needed computing resources) of individual microservices (i.e., container) to allow optimal running of the service on the one hand and to optimize the usage of the available resources on the other hand. Usually, the users or tenants have to indicate the amount of CPU and memory limit for a container running a micro-service. It happens that a container that exceeds these limits is killed or experiences a drop in performance. Accordingly, how to derive the limit to assign to a contain and configure a service resource is a challenge. On the one hand, the tenant does not clearly understand the environment in which the application will be deployed; on the other hand, the platform provider gets the application as a packaged container in which the workload is seen as a black box. In many situations, the configuration ends by using default configurations that are not appropriate for the application's requirements. Indeed, tenants naturally request a larger limit than what the application needs, which in turn, for a constrained environment (like the edge), results in resource wastage

Several works have tackled the problem of optimising application performance while reducing the amount of resources used by the latter. Most of the works propose reactive and proactive auto-scaling methods in containerized deployments. Work in [6] for instance investigated the relevance of metrics to be used as threshold metrics for scaling up and down a container. Another work [7] studied the difference between absolute versus relative metrics, i.e., metrics providing the percentage of resources used by the application from allocated resources, in microservices autoscaling. Moreover, in [8], the authors propose a Reinforcement Learning approach to autoscale microservices in the cloud. It uses two modules: the first one is a threshold-based auto-scaling algorithm deployed on Kubernetes (GKE), and the second module uses Reinforcement Learning to tune the autoscaler threshold values to obtain better threshold values to trigger autoscaling. The authors did not consider the relevance of the initial configuration.

In this paper, we will shed light on the performance of cloud-native services aiming to find solutions to correct resource misconfiguration on runtime. We run several experiments using a cloud-native platform to find solutions to overcome the resource configuration challenge. To this end, we study the behaviour of a set of representative cloud-native applications under different resource configurations and loads by measuring their performance in service response time. Then, we deduce the relationship between the service performance and the resource allocated to the workload (CPU and memory) in both absolute and relative forms. This relation allows the detection of faulty configurations and the provision of more optimal configurations for the deployed applications. The considered representative cloud-native applications are web servers and data brokers that represent vertical applications, a 5G core network service, namely Access and Mobility Management Function (AMF). Besides deriving in a generic way the threshold of CPU and memory limit from which an application does not run optimally and hence the probability of failure is high (as a container is killed if it exceeds its limit of resources), the paper's results open several perspectives. Indeed, we have also constructed different datasets using the experiment results, which may be used to run Machine Learning (ML) to predict the performances of the workloads according to their configuration.

The paper is organized as follows. Section II presents related work. Section III presents the motivation of the paper. Section IV presents the tests and the obtained results with a discussion and finally we conclude in section V.

II. Motivation

As stated earlier, most of the vertical applications are cloud-native and deployed in cloud and edge environments. However, migrating services into a new environment introduces resource allocation difficulties. First, the tenant or the application owner is not an expert in understanding the application behaviour and functioning as well as the cloud-native environment in which the application will be deployed to offer the desired services. Second, although resource over-provisioning might seem like an easy fix for this problem, this alternative introduces additional costs over long periods, which is worse when considering multiple instances of a service to deploy. Indeed, overprovisioning can lead to resource wastage which is not acceptable in an edge environment. For instance, an over-provision of an edge application will cause an overload of the Edge server, thus restricting the number of services that can run in the latter. Further, overprovision is not optimal when considering node consolidation to reduce energy consumption as the number of services to run on a node is not optimal.

On the other hand, if only horizontal scalability is available, the initial configuration of the service becomes crucial in determining the performance/resources used ratio, and a faulty configuration of the memory and CPU resources for an application can cause the container to run Out Of Memory (OOM) and experience CPU throttling or being stopped (which is harmful on the service availability). The latter can be fixed by increasing the number of instances of the application, but this will come with the cost of doubling the memory allocated to the service even though the application can perform as expected using lower memory.

Knowing the above limitations, in this paper, we investigate the behaviour of different types of services under variable load and resource limits. Our goal is to improve the initial resource configuration process and provide vertical scalability based on the limiting resource (CPU or memory). We perform tests to validate the assumption that CPU and memory relative values can be a strong indicator of the application performance and explore the cases where it is not. To this aim, we run experiments using three types of applications: 1) a Golang web server and a python web server, since web applications are widely used and deployed as the frontend of any complex service; 2) a RabbitMQ message broker, as many applications use the publish-subscribe model for communication; 3) OpenAirInterface's 5G Core Network, more specifically the AMF (Access and Mobility Management Function). For each application, we experiment with different configurations by changing the CPU limit, Memory limit, relative CPU usage (i.e. ratio), memory usage ratio, and the number of service requests received in parallel. For each configuration, we measured the latency to treat a service request (or service response time), which reflects the performance of the service in terms of Quality of Service (QoS) seen by a service consumer.

III. Conducted tests and results

In this section, we describe the performed tests and the obtained results. We have performed the tests on top of the cloud-native edge facility of EURECOM. The facility uses a Kubernetes cluster for container orchestration. We have implemented a benchmarking program that automatically deploys the application while allocating different resources for each deployment, mainly changes in CPU and memory allocated. We focused on these two metrics as they are the key resources used by the application since we did not consider access to storage or network latency and restrictions as the tested applications are run in the same cluster as the tester process. For each test, we measured the latency of the service to treat a number of requests received in parallel.

A. Testing environment

The test facility includes a Kubernetes cluster, which is deployed on top of an Intel server PowerEdge T440 with 128GB of RAM and 64 Core (Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz) with hyperthreading enabled. The cluster was bootstrapped using Kubeadm v1.20.1, and the host operating system is Ubuntu 18.04.5. All the tested applications (web servers, RabbitMQ, InfluxDB and 5G core functions) run as containers in the cluster.

The testbed, shown in Fig. 1, includes a Prometheus [1] deployment for metrics collection, a Container Infrastructure Service Manager (CISM) [9] to manage workloads (automatic

[1] https://prometheus.io/

creation and deletion of the applications pods and its necessary Kubernetes services) and software for load test such as ApacheBench [2] and RabbitMQ PerfTest [3].

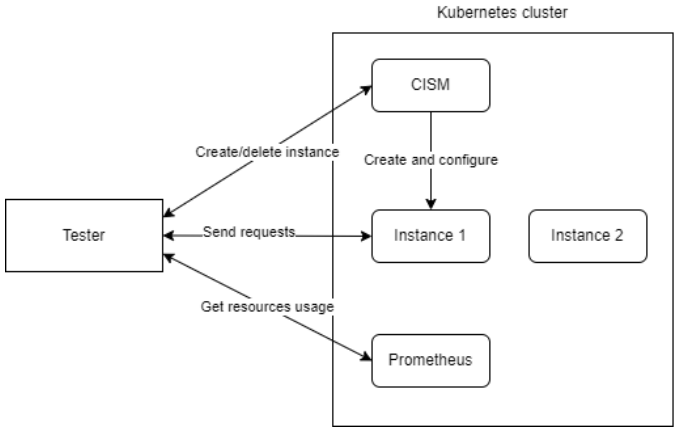

Fig. 1. The components of the testbed

B. Obtained results

We performed the tests using the following procedure. First, the test program is configured by specifying the different combinations of allocated resources. The application is then deployed with one initial configuration of resources as a container in a Kubernetes pod. Once the container is running, the application is tested with a gradually increasing load (i.e., number of service requests in parallel) until all the requests are performed, or the pod fails. While the tests are running, the tester collects resource usage from the Prometheus instance in the cluster and stores it in a file. Once all the tests are performed and the desired results are collected, the pod is deleted, and the next pod is deployed with the next configuration.

1) Web servers: We used Golang and Python-based web servers for the test. Each request to the webserver returns a video of a size 43 MB, which allowed us to overload the server with a lower number of parallel requests. We used ApacheBench, a command-line program used for benchmarking HTTP web servers, to produce high traffic. ApacheBench permits to specify the number of requests that need to be sent to the webserver and the level of concurrency, which allows parallel requests from multiple clients. We used a number of requests ranging from 100 to 1000 and a concurrency level between 1 and 100.

For allocated resources variation, we used configurations with the CPU value ranging between 0.5 and 4 CPUs with an increment of 0.5 CPU, and memory between 70 MB and 500 MB with 5MB increment between 70-100MB and 50MB increment from 100 to 500 MB. The obtained results from the two web servers were similar. The Golang based server results are shown in Fig. 2 and Fig. 3. The performance/resources trade-off is shown in Fig. 2. We can notice that under different loads, represented by the number of requests sent to the server, the more CPU the application has, the lower the response latency is. This is true for CPU allocation between 0.5 and 2 CPUs; afterward, providing more CPU does not improve the latency of the web application. Fig. 3 shows the distribution of the response time. In this experiment, we consider the relative CPU and memory, which represent the percentage of resources used from the provided limit. We notice that the higher the relative CPU (shifting vertically from one graph to the one below) is, the greater the percentage of high response times is. In contrast, the memory percentage (moving horizontally from left to right) does not change the distribution of latency values.

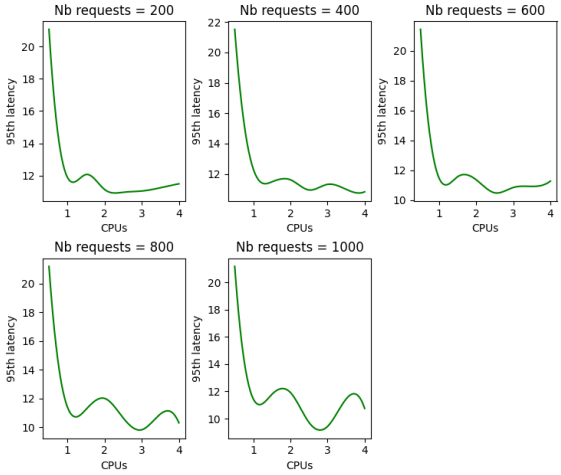

Fig. 2. Web Server's 95% latency in relation to the allocated CPU

2) 5G AMF: For the second use case, we study the performance of a 5G core network function: the AMF, which is a control plane function. Its main functions and responsibilities are registration, connection, and mobility management and access authentication and authorization. To obtain the performance of the AMF, we measure the registration time, which represents the time between the sending of the attachment requests from the UE (User Equipment) until the UE receives the authentication request.

To perform the test and generate 5G attach requests, we use my5G-RANTester [4], my5G-RANTester is a tool for emulating control and data planes of the UE and gNB (5G base station). my5G-RANTester follows the 3GPP Release 15 standard for NG-RAN (Next Generation-Radio Access Network). my5G-RANTester allows generating different workloads and testing several functionalities of a 5G core, including its compliance with the 3GPP standards. Scalability is also a relevant feature of the my5G-RANTester, which is able to mimic the behaviour of a large number of UEs and gNBs accessing simultaneously a 5G core. We deploy an OpenAirInterface

[2]https://httpd.apache.org/docs/2.4/programs/ab.html

[3]https://rabbitmq.github.io/rabbitmq-perf-test/stable/htmlsingle/

[4]https://github.com/my5G/my5G-RANTester

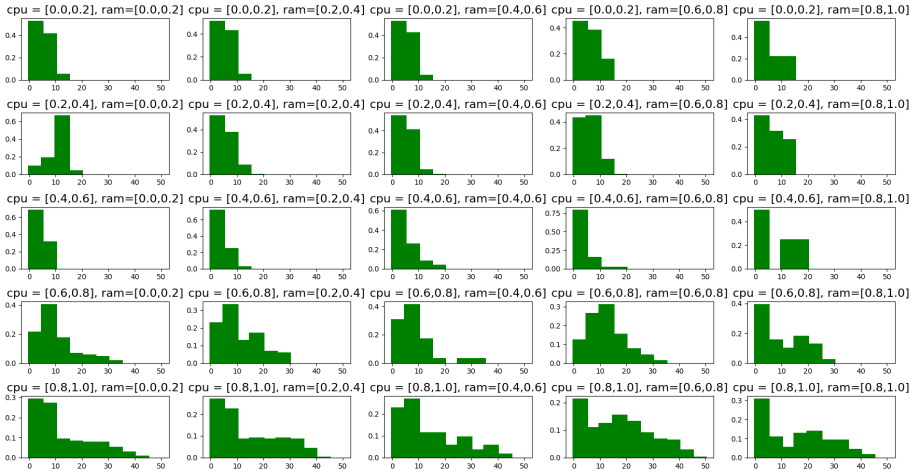

Fig. 3. Web Server's 95% latency statistical distribution

[10] core network and specify the configuration of the AMF. The explored resource allocations vary from 0.5 to 4 CPUs with an increment of 0.5 each test and memory from 256 MB to 4096 MB with an increment of 256 MB each test. The number of simultaneous registration requests that are sent to each instance varies between 10 and 400.

The obtained results are shown in Fig. 4, where the mean values for registration time (i.e., the time needed to complete the User Equipment registration to the network) in relation to the allocated CPU are displayed. We can observe that AMF performances have the same behaviour as the webserver. Indeed, we remark that the higher the CPU allocation to the AMF is, the lower the registration time is. This is valid between 0.5 and 2 CPU, after which the performance is constant. Fig. 5 shows the distribution of registration time. We see that an increase in the relative CPU results leads to high registration times.

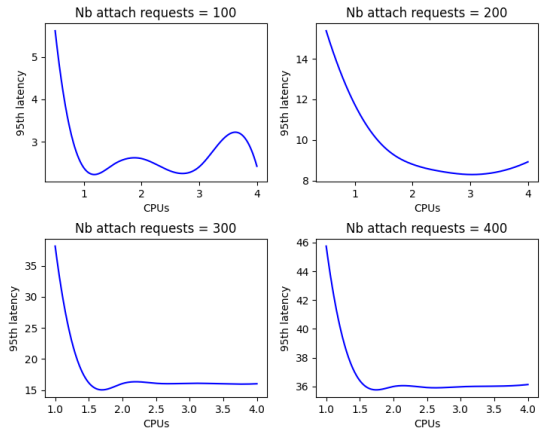

Fig. 4. AMF's 95% registration time in relation to the allocated CPU

3) RabbitMQ broker: The third microservice that we tested is a RabbitMQ messages broker. We used RabbitMQ PerfTest which is a throughput testing tool that simulates basic workloads and provides the throughput and the time that a message takes to be consumed by a consumer. For the test we used a number of producers and consumers that ranges from 50 to 500. Each producer sends messages to the broker with a rate of 100 messages per second for a period of time of 90 seconds.

Similar to the precedent tests we vary the CPU configuration of the RabbitMQ pod from 0.5 to 4 CPU, while we vary the memory from 1024 MB to 4096 with increments of 256 MB.

The results are shown in Fig. 6, we observe that the response time follows the same trend which is that the message's latency gets lower when more CPU is allocated to the broker. Fig. 7 shows that the proportion of high latencies is largest when the relative CPU utilization is between 0.8 and 1.

Interestingly, while testing the broker, and as it is a memory intensive application, even if the relative memory does not correlate with the message delay time, when the allocated memory was low (around 1024 MB) it resulted in several container restart due to an OOM signal. Fig. 8 shows the distribution of relative memory values for a memory allocation of less than 1536 MB. The values shown are the last collected memory before the failure of the pod, indeed not all failures were caused by OOM exception, but the graph shows that memory usage was approaching the set limit before the occurrence of the failure.

C. Discussion

The obtained results obviously show a strong relationship between the allocated resources and the performance of the service. The more resources the application has, the better its performance, which is valid until it reaches peak performance. The results show the effect of CPU allocation on the performance, where we clearly observed that for the same load, the response time of the service is lower when more CPU is allocated. It continues to decrease until it reaches an optimum

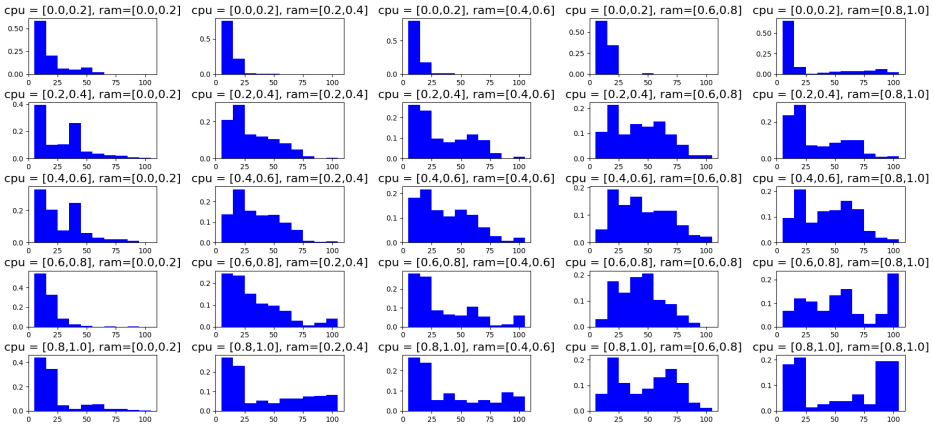

Fig. 5. 5G AMF's 95% UE registration time statistical distribution

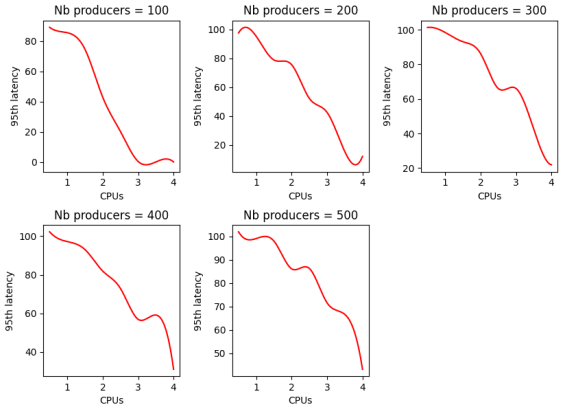

Fig. 6. RabbitMQ broker's 95% message delay in relation to the allocated CPU

performance, after which additional resources do not make the performance better, which means that a proportion of allocated resources is wasted.

Another important measure is the relative CPU. From the obtained results, we remarked that when an application's CPU usage approaches the limit allocated to the latter, the distribution of response times tends toward higher response time. Consequently, in order to detect performance degradation of applications that do not provide measures about their performances, like the latency of treated requests that is available only at the application level and difficult to derive by monitoring the infrastructure. The relative CPU and relative memory values are good indicators for (1) the application performance, where the relative CPU values indicate the service load and, consequently, the probability of occurrence of high response times. Indeed, the higher the relative CPU is, the higher the response time of a server is, whatever the type of service and the number of service requests. This finding is very important to cloud/edge providers in order to detect early misconfiguration of the service in terms of needed resources and anticipate any Service Level Agreement (SLA) issues with the tenant. Second, the relative memory consumption alerts on the occurrence of OOM signals that results in the restart of the container deploying the service, which leads to disturbing the service continuity and hence the service availability.

Finally, using the obtained results, we have implemented monitoring in our edge facility that alerts on the values of the relative CPU and memory values; by aiming to keep the relative CPU and memory values less than 0.8, which anticipate any misconfiguration of application resource. The alerts provide feedback about the application's configuration. This process helps validate the vertical resources request and assists him in finding the best configuration for the service according to the relative values of CPU and memory. If the resource usage/limit ratio approaches one, we update the Edge application with more resources to avoid performance degradation and service disturbtion.

IV. Conclusion

In this paper, we presented a study on application performance in a cloud-native and containerized environment. We have run different experiments using representative vertical applications, including a telecommunication network function. All the applications were tested under different resource configurations (CPU and memory) and loads. The obtained results provide useful insights into the behaviour of the workloads and the relation between resource usage and application performance. From those insights, we concluded that relative CPU usage is an important indicator of the relevance of the initial application resources configuration. The higher this value is, the more likely the applications will experience performance deterioration. In comparison, relative memory usage is an important indicator of the risk of occurrence of OOM errors and hence service disruption. Our future work concern the usage of the different dataset we have constructed using these

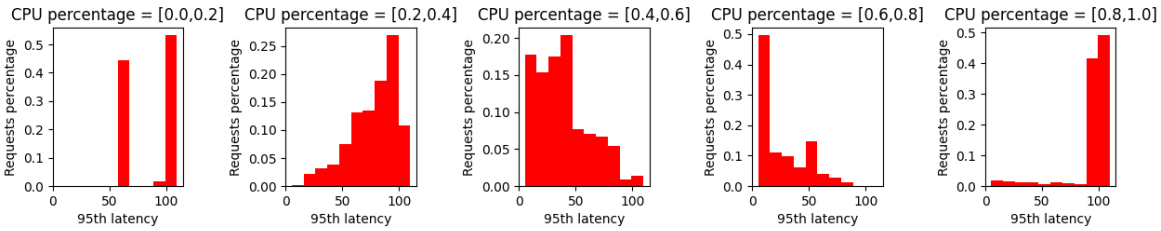

Fig. 7. RabbitMQ broker's 95% message delay statistical distribution

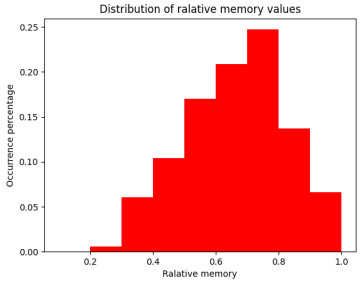

Fig. 8. RabbitMQ last recorder relative memory before pod failure

experiments to train ML models that anticipate and correct misconfiguration of application configuration to run optimally.

ACKNOWLEDGMENT

This work was partially supported by the European Union's Horizon 2020 Research and Innovation Program under the 5G!Drones project (Grant No. 857031) and MonB5G (Grant No. 871780).

REFERENCES

[1] S. Arora and A. Ksentini, "Dynamic resource allocation and placement of cloud native network services," in *ICC 2021 - IEEE International Conference on Communications, Montreal, QC, Canada, June 14-23, 2021*. IEEE, 2021, pp. 1–6.
[2] Kubernetes. [Online]. Available: https://kubernetes.io/
[3] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud amp; edge systems," 2019.
[4] S. A. et al., "Lightweight edge slice orchestration framework," in *ICC 2022 - IEEE International Conference on Communications, Montreal, Seoul, May 16-20, 2022*. IEEE, 2022.
[5] I. Afolabi, T. Taleb, P. A. Frangoudis, M. Bagaa, and A. Ksentini, "Network slicing-based customization of 5g mobile services," *IEEE Netw.*, vol. 33, no. 5, pp. 134–141, 2019.
[6] M. Gotin *et al.*, "Investigating performance metrics for scaling microservices in cloudiot-environments," 2018.
[7] E. Casalicchio and V. Perciballi, "Auto-scaling of containers: The impact of relative and absolute metrics," in *IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, 2017.
[8] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," *IEEE Access*, vol. 9, 2021.
[9] M. Mekki *et al.*, "A scalable monitoring framework for network slicing in 5g and beyond mobile networks," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 413–423, 2022.
[10] Openairinterface. [Online]. Available: https://openairinterface.org/