A Performance Comparison of Virtualization Techniques to Deploy a 5G Monitoring Platform

Ramon Perez*†, Priscilla Benedetti‡, Matteo Pergolesi*, Jaime Garcia-Reinoso†, Aitor Zabala*,
Pablo Serrano†, Mauro Femminella‡§, Gianluca Reali‡§, Albert Banchs†¶
*Telcaria Ideas, Spain †Universidad Carlos III de Madrid, Spain ‡Università degli Studi di Perugia, Italy
§Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), Italy ¶IMDEA Networks Institute, Spain

***Abstract*—The constant search for solutions to improve performance and resource efficiency in Cloud platforms has led to the introduction of new virtualization technologies and deployment paradigms. From container-based virtualization to micro virtual machines, new virtualization solutions claim to offer a performance close to bare-metal, with quick deployment and startup times. Furthermore, serverless computing has recently emerged as an alternative deployment model for Cloud workloads, providing scalability and cost reduction without requiring any additional configuration overhead from developers. In this work, we present a performance comparison of well-known virtualization technologies, *e.g.*, virtual machines or containers, together with new, serverless-oriented virtualization solutions, all of them applied to deploy a real monitoring architecture for multi-site 5G platforms. The analysis shows that serverless technologies do not achieve the same performance than well-known virtualization technologies, evaluating as a result the suitability of these new virtualization technologies for production environments, extracting useful conclusions to lay the ground for future work related to this novel paradigm.**

***Index Terms*—Performance comparison, virtualization techniques, serverless, 5G networks, monitoring and data collection.**

I. Introduction

Today's Cloud platforms allow to rapidly deploy new services, extending existing applications with additional resources within seconds. However, the increasing prevalence of the microservice paradigm creates a new demand for low-overhead virtualization techniques [1]. Indeed, while virtual machines are one of the most popular virtualization approaches for deploying services on the Cloud, this technology is inefficient because it relies on overprovisioning and long startup times. Another widespread approach in virtualization is represented by containers, which wrap applications with their entire runtime environment and execute them on the host as processes, therefore containers get isolated filesystems and resources while sharing the same host system kernel. This property enables containers to start an order of magnitude faster than virtual machines, with increased resource efficiency. Nevertheless, the usage of a shared kernel can lead to some security and isolation issues, along with long cold startup latencies [2][3].

In order to find virtualization solutions for fast provisioning, resource efficiency and security, new lightweight virtualization platforms have been gradually introduced in recent years, such as unikernels, micro virtual machines (microVMs) and minimal hypervisors [4]. Moreover, the introduction of the serverless deployment model, relying on lightweight virtualization technologies, allows to adjust the amount of resources allocated for a given workload with a finer level of granularity, achieving a better performance profile by using functions that are triggered only when needed. As a result, in the near future, serverless computing could become a possibility for implementing a wide range of flexible and self-optimizing applications and communication services, as well as for implementing optimized virtual network functions [5].

Nevertheless, some aspects related to the real performance of the serverless architectures and new lightweight virtualization techniques are still open, such as the comparison with traditional virtualization techniques, in order to confirm whether these new solutions are really more efficient in terms of resource consumption, or also the verification that not only serverless-oriented microservices, but also standard components executed by using serverless paradigm, achieve a better performance profile than with traditional technologies.

With these topics in mind, this paper presents a full performance evaluation of different deployments of a given specific-purpose platform, each of them making use of a specific virtualization technique. The main objective is to compare these techniques between them in terms of specific performance parameters, in order to position new virtualization approaches and serverless paradigm in the state of the art.

To this end, the system evaluated in this comparison is the 5G EVE[1] Monitoring framework, which allows the distribution and consumption of metrics and KPIs for 5G multi-site platforms [6]. For the comparison, this platform has been deployed by using the following virtualization techniques: (1) the case of not using virtualization, so bare-metal servers are directly used, (2) virtualization based on *KVM*, (3) containerization with *Docker* with *runc* as runtime, (4) automatic deployment of containers with *Kubernetes*, using *containerd* as runtime, and (5) the usage of *Kata Containers* with *Firecracker* or *qemu* as hypervisors, being *Kata Containers* and *Firecracker* two highly adopted new virtualization technologies, suitable for the serverless paradigm [7][8].

To do this, the rest of the paper is organized as follows: **Section II** presents the background related to the new technologies introduced in this work, together with other comparatives already made in the state of the art. **Section III** presents an overview of the Monitoring platform under study, describing

[1]https://www.5g-eve.eu/

its building blocks and discussing the suitability of using these novel virtualization techniques on it. **Section IV** delves into the performance evaluation analysis of each virtualization technology used to deploy the Monitoring platform with (*i*) a single-server deployment, and (*ii*) applying *Kubernetes* for horizontal scaling to improve the results, in terms of performance parameters. Finally, **Section V** concludes the paper and presents our future work.

II. Background and Related Work

Lightweight virtualization technologies try to address both resource efficiency and security through isolation. *Unikernels*, which are minimalistic runtime environments running on top of virtual hardware abstractions, are proposed as an alternative to containers, since they provide more security and are often faster. However, by removing the concept of *process* from their monolithic appliances, unikernels gain less flexibility and applicability since dynamic forking, which is a basis for the common multi-process abstraction of conventional UNIX applications, is not supported. Moreover, the unikernels' peculiarity of running their applications directly in the kernel ring, *i.e.*, in the same address space, could also entail some security issues, since privilege separation is impractical and dangerous kernel functions could be invoked from applications [9][10].

Kata Containers [7] is another lightweight virtualization technology that encapsulates containers processes and namespaces into microVMs, for enhanced isolation. This approach is meant to bring together the speed of containers and the security offered by virtual machines into a single product that expects to propose a two-layer system-wide isolation for improving the security capabilities of containerized components.

Indeed, the *Kata Containers* runtime (*kata-runtime*) is compatible with the Open Container Initiative (OCI) runtime specification and the *Kubernetes* Container Runtime Interface (CRI). Therefore, it works seamlessly with *Docker* engine and *Kubernetes* kubelet, deploying a *KVM* virtual machine for each container or pod created with multiple hypervisors supported, even including specific hypervisors adapted to the serverless trend. An example of this is *Firecracker* [8], an open-source, minimalist Virtual Machine Monitor that uses *KVM* virtualization infrastructure to create and manage microVMs, written in *Rust* [11]. In a way similar to unikernels, Firecracker is built with minimal device emulation that enables faster startup time and a reduced memory footprint for each microVM.

With regards to the related work, focusing on performance comparisons between different virtualization techniques, various works have been published, comparing virtual machines, containers or unikernels, as presented in [1][12][13]. Some conclusions can be extracted from them, such as the small performance difference between containers and hypervisor-based virtualization, and also between containers and unikernels. Nevertheless, from these works, further comparisons with new lightweight technologies focused on security such as microVMs are not considered. Consequently, this paper extends previous works on virtualization techniques evaluation by introducing a performance comparison of modern lightweight virtualization technologies, such as microVMs and modular hypervisors with traditional KVM virtualization and containers.

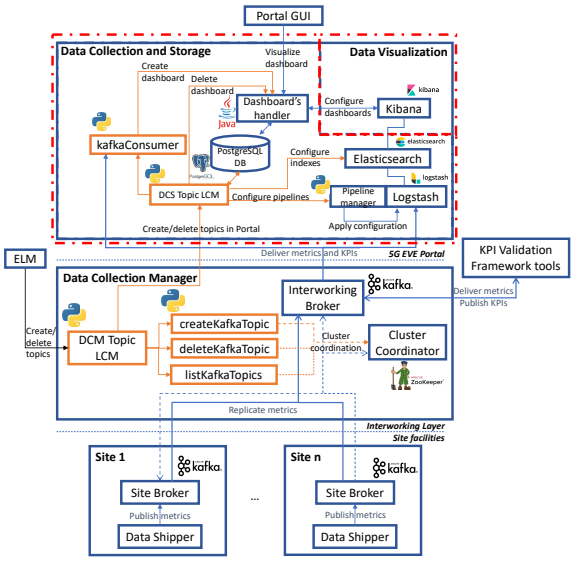

Fig. 1. Monitoring framework implementation based on microservices in a multi-site environment.

The performance comparison presented in this paper has been conducted on the Monitoring platform introduced in [6]. In that paper, apart from describing the design and implementation of this multi-site, multi-tenant Monitoring system, it also validates such implementation based on *Docker* containers deployed on virtual machines with a preliminary performance evaluation analysis. As a result, some interesting conclusions can be extracted from that work, such as the CPU-intensive behavior of the platform, or the loss of performance due to a saturation effect caused by the technology itself. All these conclusions, together with the testing procedure defined in that analysis, have been used as the basis for this work.

Taking into account the reviewed background and related work, it can be concluded that a performance comparison and suitability analysis of the lightweight virtualization solutions, together with mature technologies such as virtual machines or containers, is still not present in literature. Consequently, the evaluation presented in this paper, which is the main contribution of this work, intends to fill in this gap detected in the state of the art.

III. 5G EVE Monitoring Platform Overview

For the sake of completeness, the implementation of the Monitoring framework studied in [6] will be reviewed, highlighting the building blocks that compose each component of the architecture. In this way, a microservice-based architecture of the Monitoring platform is depicted in Figure 1.

This architecture extends the latest reported implementation of both the *Data Collection Manager* (DCM) [14] and the

Data Collection and Storage-Data Visualization (DCS-DV) [15] entities. This higher level of granularity in the description of the architecture allows to better understand the workflow followed by the monitoring traffic in both the control and data plane, and also lays the ground for integrating serverless mechanisms and frameworks in specific blocks where the use of these technologies makes sense.

In this new architecture, the general-purpose building blocks introduced in the Monitoring framework designed in [6] are easily identified. For example, the complete *Broker System* is divided in 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*[2], and are coordinated and orchestrated by the *Cluster Coordinator*, which is based on *Apache ZooKeeper*[3].

Additionally, the *Data Collection Manager* has other blocks (remarked in orange in Fig. 1) that fit in the *Metrics Management* entity from the general design, as they are related to configuration processes in the control plane. The main block from this workflow is the *DCM Topic LCM*, which is in charge of managing the life-cycle of the topics to be managed by the platform. The operations to be performed on the topics are triggered by an external entity, which is the *Experiment Lifecycle Manager* (ELM in Fig. 1), and these operations are represented by the three remaining blocks in the DCM: *createKafkaTopic*, *deleteKafkaTopic* and *listKafkaTopics*.

Continuing with the *Metrics aggregation* and the *Monitoring/Results collection tools* entities from the general design, they are included in the DCS-DV architecture. In terms of the data plane, this component, which relies on the *Elastic Stack*[4], manages the metric/KPI collection, indexing and visualization, and it is separated logically in two main blocks:

- The *Data Collection and Storage* component (DCS), which firstly aggregates and collects the metrics and KPIs to which this component is subscribed through *Logstash*. Then, it provides data persistence, searching and filtering capabilities for obtaining the useful data to be monitored during the experiment thanks to *Elasticsearch*.
- The *Data Visualization* component (DV), in charge of enabling the monitoring of the progress of the experiment in terms of that monitoring data displayed through *Kibana*. For this purpose, a set of dashboards are created, presenting the graphs related to each metric or KPI.

The DCS-DV control plane[5] is mainly based on the *DCS Topic LCM*, which automates the process of correctly configuring both *Logstash* (interacting with the *Pipeline manager* entity) and *Elasticsearch*, and also triggering the dashboards' creation (through the *kafkaConsumer* entity, ensuring that a dashboard is only created when data is received in the corresponding topic) and removal (through the *Dashboard's handler* entity, an application which directly interacts with *Kibana* and the 5G EVE *Portal GUI* for offering the dashboards to the verticals).

In fact, all the building blocks that belong to the *Metrics Management* entity have something in common: they do not need to be running all the time. This feature may have an impact on performance in the platform, as there are components that are consuming system resources while they are idling most of the time. And not only this: if moving to a Cloud platform, applying a pay-per-use model, this also involves a waste of money in resources that are scarcely used.

Consequently, these kind of building blocks are perfect candidates to be migrated to serverless technologies, so that they are only used when required. To do this, a serverless framework would be needed, abstracting the technical features of the infrastructure to make the process of designing, developing and deploying the serverless functions easier [16]. This platform would manage the lifecycle of the serverless functions, thus instantiating the resources needed to execute the functions, and then releasing them when it finishes its execution. However, this work is out of the scope of this paper.

[2]https://kafka.apache.org/

[3]https://zookeeper.apache.org/

[4]https://www.elastic.co/es/elastic-stack

[5]The blocks involved in this plane, which also fit in the functions carried out by the *Metrics Management* entity, are also remarked in orange in Fig. 1.

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

IV. PERFORMANCE EVALUATION

The performance comparison between virtualization techniques has followed the testing process described next.

A. Testbed setup

For this evaluation process, two physical servers[6] based on *Ubuntu Server 20.04* have been used. Their hardware specifications are fully described in Table I. Although the CPU model of each server is slightly different, they do not present a big difference in terms of performance[7].

In **Server #1**, the *Data Collection Manager* component from the Monitoring platform[8] has been deployed for each virtualization technique studied. In particular, the only subcomponents used from the DCM for the tests are *Kafka* and *ZooKeeper*. Moreover, a CPU collector script based on the *mpstat* command is used for measuring the CPU consumption.

The testbeds deployed in Server #1 to evaluate the technologies under study are the following:

[6]Time synchronization is achieved by using the NTP protocol.

[7]https://gist.github.com/TheWall89/01688a7f448d8403da7798bcdf0185bc

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

- ***Bare-metal* testbed**, using directly Server #1 without any virtualization technique. In this case, *Kafka* and *ZooKeeper* are directly installed as Linux services.
- ***KVM* testbed**, deploying *Kafka* and *ZooKeeper* in Server #1 in a specific Ubuntu virtual machine[9].
- ***Docker* testbed**, using *runc* as container runtime (the default one) for deploying *Kafka* and *ZooKeeper*.
- ***Kubernetes* testbed**, using *MicroK8s*, a *Kubernetes* minimal production version[10], with *containerd* as runtime, in order to deploy a *Kafka* and a *ZooKeeper* pod.
- ***Kata* testbed**, which is similar to the *Docker* testbed, but using a customized runtime that can be either *Kata+Firecracker* or *Kata+QEMU* for the evaluation of serverless tools.

With regard to **Server #2**, a *Kafka* publisher based on *Sangrenel*[11] is used for the obtention of the performance parameters under study.

B. Test Plan

In general terms, the same assumptions commented on [6] also apply to this evaluation process, having a maximum number of six experiments running simultaneously on the Monitoring platform, considering that one experiment implies the creation of 20 topics in the system, with a concurrent publication rate of approximately 102,4 $Mbps$ per experiment.

Delving into the particular tests to be executed on each testbed, they consist on the execution of an experiment in the Monitoring platform, publishing data in *Kafka* at a given data rate (depending on the number of topics present in the platform) with *Sangrenel*. Two types of tests are carried out:

- **Single server tests** (reported in Section IV-C): this set of tests implies the execution of tests with a single instance of *Apache Kafka* running in the Monitoring platform.
- **Horizontal scaling tests** (reported in Section IV-D): this configuration is an extension of the work already done in [6] in terms of vertical scaling mechanisms. It is also an enhancement of the previous test type, in which it is also tested the case of having two *Kafka* instances running in parallel, processing simultaneously the traffic received by the platform. The idea is to compare the performance achieved in this configuration with the one obtained in the single server case. These tests are only performed in the *Kubernetes* testbed, as it is currently the more mature technology that enables the orchestration of multiple instances of the same service.

These are the parameters that allow to fully define the tests:

- **Design parameters:** they are related to input data to the system in order to configure properly the Monitoring platform for the tests. We can distinguish between:
  - **Fixed:** for each experiment, there are 8 topics managing messages with a size of 100 B, 8 topics managing 1 KB messages, 2 topics managing 100 KB messages and 2 topics managing 1 MB messages. The test duration is 5 minutes[12], and the number of test repetitions is fixed to 10 repetitions.
  - **Variable:** there are 20 topics per experiment, varying from 1 to 6 experiments. This parameter determines the throughput received by the Monitoring platform.
- **Performance parameters:** these are the parameters measured during the execution of the tests, which can be:
  - **CPU consumption:** measured on Server #1 server 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 must be turned off.
  - **Batch write latency:** the time spent until receiving an ACK message from the *Kafka* broker.
  - **I/O message rate:** the received throughput divided by the publication rate.

C. Single-Server Evaluation

For the first set of tests executed on each testbed, in which only one *Kafka* broker is present, the following results are obtained, in terms of the different performance parameters defined in Section IV-B.

In the case of the CPU consumption, whose results for all testbeds can be found in Figure 2, the saturation effect observed in [6] is present. This means that the CPU consumption increases its value when increasing the number of experiments deployed until reaching a hard limit (*i.e.*, the saturation point), obtaining around 27% for the *bare-metal*, virtual, *Docker* and *Kubernetes* testbeds, and 6-8% for both *Kata* testbeds. Comparing all the scenarios, the following tendencies are observed:

- The *bare-metal*, *Docker* and *Kubernetes* testbeds present a similar tendency[13], starting with a value 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 than the previous case.
- On the other hand, both *Kata* options saturate much sooner, with a lower throughput received (*i.e.*, less than 102,4 $Mbps$). As a result, the CPU consumption remains constant with a lower value compared to the other scenarios, making also an impact on the other parameters under study. In separate tests, running experiments with a total throughput less than 102,4 $Mbps$, the saturation point is determined for a throughput between 25,6 (5 topics) and 51,2 (10 topics) $Mbps$.

[9]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.

[10]The Kubernetes distribution does not affect the performance of the container runtime.

[11]https://github.com/jamiealquiza/sangrenel

[12]This test duration, together with the selected number of repetitions, allows to obtain significant and stable results in statistical terms.

[13]In the case of the analysis from [6], where containers are deployed wihin VMs, the results are similar to the *KVM* testbed from this analysis, as containers adapt their performance to the environment in which they run.

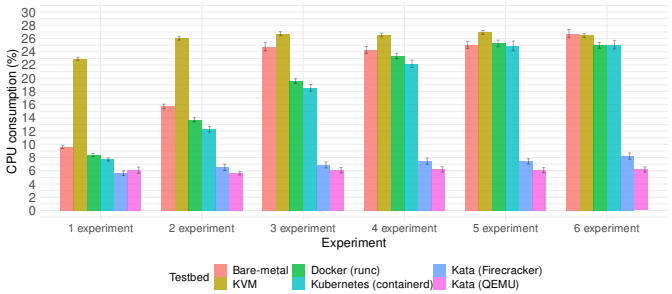

Fig. 2. CPU consumption evolution for all the testbeds of the Monitoring platform (100 B messages, 95% confidence interval).

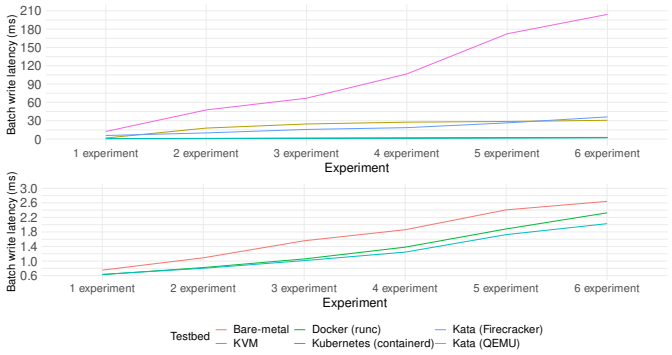

Fig. 3. (Top) Batch write latency evolution for all the testbeds of the Monitoring platform, (bottom) also detailing the results for the testbeds with lower values (100 B messages).

The batch write latency, whose related results are presented in Figure 3, also increases its value when the number of experiments deployed becomes higher. For this parameter, three different trends are also observed, but with different implications compared to the CPU consumption:

- Again. the best results are observed for the *bare-metal*, *Docker* and *Kubernetes* testbeds, with a batch write latency value lower than 3 ms in the worst case (*i.e.*, with 6 experiments deployed).
- In the second place, it comes the *KVM*, and also the *Kata+Firecracker* scenario, with one order of magnitude more than the previous case (around 40 ms in the worst case). This happens because the hypervisor's access to disk is different than with containers (and obviously with a direct access to the disk, which is the case of the *bare-metal* scenario), as containers share resources with the host (*i.e.*, directly the bare-metal server).
- Finally, the worst results are obtained for the *Kata+QEMU* scenario, having another order of magnitude more compared to the previous case (around 200 ms in the worst case) due to a *heavy* packet loss process experienced, as it is observed in the analysis of the I/O message rate.

And finally, the evolution of the I/O message rate, according to the results presented on Figure 4, also depends on the saturation effect experienced in *Kafka*: when it appears,

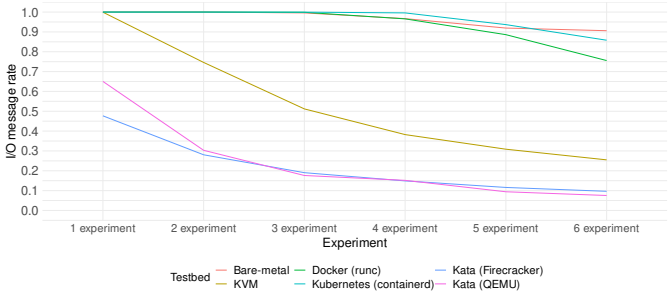

Fig. 4. I/O message rate evolution for all the testbeds of the Monitoring platform (100 B messages).

packets start to be lost, causing a lower I/O message rate when increasing the number of experiments executed (*i.e.*, the throughput received by the platform). In this way, the three tendencies observed for the CPU consumption are also repeated here with a clear correlation between results:

- First of all, the *bare-metal*, *Docker* and *Kubernetes* testbeds present the same tendency and the highest values possible, with an I/O message rate of around 0,75 in the worst case (*Docker* testbed with 6 experiments deployed). According to the moment in which the I/O message rate starts to fall, it is confirmed that the saturation point seems to be between 3 (for *bare-metal* and *Docker*) and 4 (for *Kubernetes*) experiments deployed.
- Then, it comes the *KVM* testbed, in which the saturation point is produced with 2 experiments, as commented in the CPU consumption analysis, and achieving a value of around 0,25 in the worst case. This trend is, in fact, the one observed in the analysis done in [6], as containers are used in a VM as host.
- And finally, both *Kata* options have a poor performance, starting with around 0,65 (*QEMU*) and 0,5 (*Firecracker*) for 1 experiment and falling to less than 0,2 for 6 experiments.

D. Horizontal Scaling Evaluation

Next, we consider a scenario where we have multiple *Kafka* brokers running at the same time, achieved by using *Kubernetes* services and deployments instead of pods. This way, *Kubernetes* is able to automatically manage the number of *Kafka* instances running on the platform (in this case, limited by two).

For the test cases, the same procedure explained on Section IV-B are followed, obtaining better results than in the single-server evaluation case for all the performance parameters under study. For example, in the case of the batch write latency, presented in Figure 5, the values obtained for the two-broker scenario are always lower than the single-broker case, also with a lower slope. For the worst case, the latency is reduced to the half approximately, achieving around 1 ms.

V. Conclusions and Future Work

From the analysis presented in this work, it can be concluded that the different testbeds can be classified in three

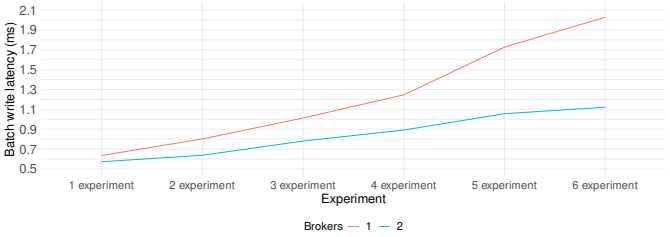

Fig. 5. Batch write latency evolution for the *Kubernetes* testbed with one or two *Kafka* brokers (100 B messages).

groups, according to the results obtained: in the first group, it appears the *bare-metal*, *Docker* and *Kubernetes* testbeds, which have a similar performance profile and the best results, making sense to use containers over bare-metal servers for lightweight software. This confirms the good result expected for container-based solutions, based on the analysis made in [13], but its versatility is paid in terms of security. Despite this, these security issues can be solved with the introduction of novel solutions like *Kata Containers* or *Firecracker*, which intend to improve security management while following the same philosophy of lightweight deployments.

The second group is mainly composed by the *KVM* testbed, offering a worse performance (probably due to not having direct access to the disk resources), but with reasonable values to use it in production environments. And in third place, the results for both *Kata* testbeds, compared with the others, are the worst ones, going to scenarios with less than 20 topics to confirm that these technologies are working properly.

As a result, there are clear evidences that, based on the issues reported and the problems found during the tests, it can be supposed that this kind of lightweight virtualization technologies are not mature yet for using them in production environments managing data-intensive workloads. In particular, one of the problems found is related to disk access, a highly required capability by *Kafka* to work properly. In fact, this issue has been already covered in the state of the art with studies like [17], pointing out that *Kata Containers* are not really efficient in terms of memory consumption and speed, while still being a good deployment choice in security-sensitive multi-tenant environments.

Of course, this evidence cannot be extrapolated at all for all cases from the results obtained, but this can be done, at least, for the testing process followed. And also, it is expected that the technology will be optimized somehow in the near future, eventually allowing them to achieve a better performance profile, enabling then the introduction of this technology in production-oriented systems such as the 5G EVE Monitoring platform.

Finally, preliminary horizontal scaling techniques in the *Kubernetes* testbed have also been analyzed, confirming that the performance profile improves considerably and that could be a candidate for the extension and improvement of the platform in future releases. For example, proposing the use of *Kubernetes* for serverless deployments based on *Kata-Firecracker*, combining the promising features offered by these serverless techniques and the good performance results obtained by *Kubernetes* in this work, so that platforms like the 5G EVE Monitoring framework can leverage on serverless technologies to benefit from these improvements.

Moreover, note that the performance evaluation presented here only evaluates *Kafka* and *ZooKeeper*, but it skips the analysis of the possible serverless functions proposed for the Monitoring platform in Section III. This is, in fact, the main future work for this research topic, and this performance evaluation of these functions is currently under study, taking advantage of the well-known open-source serverless platform *OpenFaaS* [16] to create and deploy some modules of the Monitoring platform in a serverless way on top of *Kubernetes*.

ACKNOWLEDGEMENTS

This work has been partially supported by the Community of Madrid (grant IND2017/TIC-7732) and the EC H2020 5G-PPP 5G EVE project (grant 815074).

REFERENCES

[1] M. Plauth *et al.*, "A Performance Survey of Lightweight Virtualization Techniques," in *Service-Oriented and Cloud Computing*, vol. 10465. Springer International Publishing, 2017, p. 34–48.
[2] "CVE Details: CVE-2019-5736," Jun. 2019. [Online]. Available: https://www.cvedetails.com/cve/CVE-2019-5736/
[3] "Exploiting CVE-2017-5123," Nov. 2017. [Online]. Available: https://reverse.put.as/2017/11/07/exploiting-cve-2017-5123/
[4] J. Chen, "Making Containers More Isolated: An Overview of Sandboxed Container Technologies," Jun. 2019. [Online]. Available: https://bit.ly/2ZcRBOR
[5] P. Aditya *et al.*, "Will Serverless Computing Revolutionize NFV?" *Proceedings of the IEEE*, vol. 107, no. 4, pp. 667–678, 2019.
[6] R. Perez *et al.*, "A Monitoring Framework for Multi-Site 5G Platforms," in *EuCNC 2020*, 2020, pp. 52–56.
[7] A. Randazzo and I. Tinnirello, "Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way," in *IOTSMS 2019*, 2019, pp. 209–214.
[8] A. Agache *et al.*, "Firecracker: Lightweight Virtualization for Serverless Applications," in *17th USENIX NSDI*, Santa Clara, CA, Feb. 2020, pp. 419–434.
[9] Y. Zhang et al., "KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization," in *USENIX ATC 18*, Boston, MA, 2018, pp. 173–186.
[10] J. Talbot et al., "A Security Perspective on Unikernels," Nov. 2019. [Online]. Available: https://arxiv.org/abs/1911.06260
[11] R. Jung *et al.*, "RustBelt: Securing the Foundations of the Rust Programming Language," in *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017.
[12] I. Mavridis and H. Karatza, "Lightweight Virtualization Approaches for Software-Defined Systems and Cloud Computing: An Evaluation of Unikernels and Containers," in *SDS 2019*. IEEE, Jun 2019, p. 171–178.
[13] R. Morabito, J. Kjallman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *2015 IEEE International Conference on Cloud Engineering*. IEEE, Mar 2015, p. 386–393.
[14] 5G-EVE, "Second implementation of the interworking reference model," Deliverable D3.4, Jun. 2020. [Online]. Available: https://zenodo.org/record/3946323#.YCKm151KhH4
[15] ——, "Report on benchmarking of new features and on the experimental portal (2nd version)," Deliverable D4.4, Jun. 2020. [Online]. Available: https://zenodo.org/record/3946283#.YCKm2Z1KhH4
[16] K. Kritikos and P. Skrzypek, "A Review of Serverless Frameworks," in *IEEE/ACM UCC Companion*, 2018, pp. 161–168.
[17] V. Aggarwal and B. Thangaraju, "Performance Analysis of Virtualisation Technologies in NFV and Edge Deployments," in *IEEE CONECCT 2020*, 2020, pp. 1–5.