Microservice performance in Container- and Function-as-a-Service architectures

Claudia Canali†, Riccardo Lancellotti†, Pietro Pedroni*
† *Department of Engineering "Enzo Ferrari"*, *University of Modena and Reggio Emilia*, Modena, Italy
* *Department of Physics, Informatics and Mathematics*, *University of Modena and Reggio Emilia*, Modena, Italy
{claudia.canali, riccardo.lancellotti, }@unimore.it, pedroni.pietro.96@gmail.com

***Abstract*—Function-as-a-Service (FaaS) is a new cloud-based computing model that promises a more cost-efficient deployment of microservices with respect to other cloud paradigms, like Container-as-a-Service (CaaS). However, requests served under a FaaS approach often experience a cold start condition, that occurs when the execution of an inactive function occurs for the first time and a container environment has to be set up afresh. In such cases, performance deteriorates and response times increase. This paper proposes an analysis of the performance of the Function-as-a-Service model for two single offered microservices. Specifically, we carry out a performance evaluation of the Function-as-a-Service model, implemented through OpenWhisk, using as a baseline for comparison the Container-as-a-Service approach, implemented with Docker. Our analysis focuses on metrics related to the response time and to the usage of main server resources such as CPU and memory. For the performance comparison, we exploited two different microservices based on face recognition and image conversion, respectively, in order to evaluate the performance over popular and modern kinds of services included in artificial intelligence and multimedia applications.**

***Index Terms*—Function-as-a-Service, Cloud Computing, Performance, Microservice**

I. Introduction

The Function-as-a-service (FaaS) model, also called Serverless computing [1], has been defined as the next phase of cloud computing thanks to its capabilities of allowing an efficient development and deployment of applications without the need to manage the underlying infrastructure [2], [3]. Serverless computing platforms, indeed, enable developers to focus on code and business logic, without the overhead of scaling and provisioning the system. One important advantage of this paradigm is the exploitation of a pay-as-you-go pricing model, where users are only charged for what they actually use [4]. For this reason, the Function-as-a-Service model is typically tailored for short-running microservices that are called on-demand.

The advantages of this emerging model make FaaS a good candidate for being increasingly adopted. However, the FaaS model opens up to novel considerations that must be taken into account, such as the startup overheads of container-based environment [5]. Indeed, in order to execute a microservice, the corresponding execution environment should be brought up and this frequently involves a cold start condition, when the container is started from scratch [6], [7]. Even if containers are much faster than VMs, their cold start time can still be in the comparable with the typical function's execution. Hence, cord start can have a significant impact on response times [8].

Some papers in literature have evaluated the behaviour of the FaaS model [9]–[11]. In [9], the authors evaluate the applicability of the FaaS approach to compute- and data-intensive scientific workflows and discuss possible ways to repurpose serverless architectures for execution of scientific workflows. The study in [11] carries out an assessment to study the suitability of AWS Lambda as a platform for the execution of High Throughput Computing jobs, executing MapReduce jobs on AWS Lambda, using Amazon S3 as the storage backend. These studies, as most of the literature, mainly focus on hosted platforms that are fully managed by a public cloud provider offering FaaS services, such as Google Cloud Functions [12], Microsoft Azure [13] and AWS Lambda [14]. Specific performance tuning and additional features aimed to reduce the impact of container startup costs in such systems are documented [15]. This focus clearly emerges by the multivocal review in [10], covering 112 studies from academic and industrial literature. Hence, the underlying platform remains somehow unclear because it is not under the control of the customer running the experiments. The study in [16] presents SEBS (Serverless Benchmark Suite), that allows to specify workloads, monitoring and evaluation tasks. The FaaS abstract model of implementation supports the benchmark applicability to commercial vendors (e.g., AWS, Azure and GCF) and evaluates metrics as time, CPU, memory and I/O. However, it does not support testing on open-source platforms that may be deployed in a private cloud environment.

Few studies consider private cloud and open source software, such as OpenWhisk. However such research typically propose a platform modification to optimize the provisioning policies depending on the workload [17] or for general performance optimization [18].

This paper proposes a performance evaluation of the Function-as-a-Service model, implemented using an open-source platform on a private cloud, and a comparison with the Container-as-a-Service approach. The serverless architecture is implemented by using open source Apache OpenWhisk [19], while the CaaS approach is realized with the Docker containerization platform [20]; the performance evaluation takes into account the main metrics of response time, CPU utilization and memory usage. Specifically, we test the performance for two common microservices involved in artificial intelligence and

multimedia applications: face recognition and image conversion. In our experiments we consider two operating conditions for the micro-services: sequence of requests and requests with a long inter-arrival interval, corresponding to warm- and cold-start of micro-services. Our results show how the Function-as-a-service approach actually presents a significant hidden cost that should be carefully considered for services characterized by sporadic invocations and low response times requirements. It is worth to note that the performance evaluation presented in this paper can provide real data about response time and resources utilization of microservices over FaaS technologies. We believe that this contribution could represent a useful input for scientific studies that exploit simulations to evaluate FaaS-based systems, such as [21], [22].

The rest of this paper is organized as following. Section II describes the architecture and the services used to test the performance of the considered technologies, while III presents the experimental results. Finally, Section IV concludes the paper with some final remarks.

II. Testing architecture

To carry out our performance evaluation, we rely on a testing architecture that is presented in this section. We first describe the technologies used to evaluate and compare FaaS and CaaS approaches, Then, we describe the microservices implemented to test the performance.

A. Testbed Deployment

In our performance evaluation we rely on the following software packages: Docker, OpenWhisk and Prometheus. Docker and OpenWhisk are used in our experiments for the CaaS and FaaS models, respectively. Prometheus, instead, is used on booth CaaS and FaaS platforms to collect data on the resource utilization.

Docker Container

Containers are an alternative to classic virtual machines and represent the solution to virtualization from a software point of view. Containers encapsulates the code, dependencies and environment of the entire application, thus guaranteeing a great portability over different underlying systems. Containers can be used separately or can be aggregated to create larger and more complex applications.

In this paper, we consider Docker [20] as the leading container management technology. Docker key aspects are:

- Limited overhead. As the containers installed on a machine share the operating system kernel, therefore, their startup is quick and requires little CPU and RAM. The disk space occupied by the containers is also reduced thanks to sharing of common files among different containers.
- Simple management, since containers by their nature can be created and destroyed quickly and with the use of few resources.
- Independence. As containers, which are based on standards, can be moved from one operating system to another because they are distributed in a standard format (Docker Image), which can be read and executed by any Docker server.
- Safety. As the containers are independent from the others and the problems affecting one of them do not affect the rest of the machine.
- Support for permissions. In order to use Docker it is necessary that the kernel supports the capabilities, used to perform operations that require certain privileges and system calls.

OpenWhisk

Apache OpenWhisk [19] is a flexible Servless opensource platform developed by the Apache Software Foundation. This platform is entirely based on the Function-as-a-Service paradigm where functions are typically performed in response to events. Functions (called Actions) are autonomous and independent of external events. Actions can be written in any programming language for which a compiler or interpreter exists and the data type for the input and output of an Action must be a dictionary data structure, where the keys are strings and the values are JSON subjects. This standard data type allows the concatenation of multiple functions. In terms of events, OpenWhisk manages a Trigger and Rule system: Triggers are basically event listeners; Rules correspond to the connection between a certain Trigger and an Action. Every time an event from the external environment (Trigger) is intercepted, a rule (Rule) is applied to activate a certain function (Action). The types of events are highly heterogeneous. A non exhaustive list of events includes: changes in databases, interactions from HTTP requests, and API invocation.

Due to the serverless nature of OpenWhisk, a function that is activated for the first time causes a condition called Cold Start. Such condition causes a delay because the corresponding container is not ready for execution. On the other hand, subsequent executions (invoked within a time frame such that the container providing the function is not switched off) will be activated through a so-called Warm Start, for which the activation time is negligible. The output of each Action is provided in JSON format. For storing function sources, execution states, triggers, rules, namespaces and other data, Apache CouchDB is used by default, a NoSQL database that uses JSON to store data and executes queries in MapReduce. All the components of OpenWhisk make up a microservices structure that can be enclosed within a Docker Container.

Prometheus

Prometheus is a monitoring tool that stores the metrics of the linked systems in a proprietary timeseries database. As previously pointed out, Prometheus is used to collect data about the resource utilization of the systems under tests. The data can be recovered and aggregated using the specific language PromQL (Prometheus Query Language), through which queries are sent to the DB and the desired results are observed. Each endpoint, or the services thus defined by Prometheus, exposes its own metrics based on a “Single Web

page for all metrics" approach. Prometheus then retrieves all the metrics for which it was configured through a pull. To all intents and purposes, therefore, the services will only have to expose their list of metrics via REST. Many programming libraries are provided for writing in the correct format. A metric contained in timeseries is made up of a key-value pair: metric name (unit of measure) label1 = value1, label2 = value2, ... value (timestamp) We choose Prometheus for its simplicity in collecting metrics without the need to install agents on the endpoints.

B. Microservices

The services used during testing are two, namely face recognition and image conversion. The microservices used for the analyses are performed inside suitable containers, created ad hoc. OpenWhisk, in addition to the normal function as a service execution, supports the execution scripts by relying on an environment created in Docker and in this regard this environment, which corresponds to the container, is the same for both OpenWhisk and Docker. Each container is based on a Docker Image, available on Docker Hub, structured on python version 3.9. We have therefore generated a Docker File that includes all the libraries necessary for the execution of the two microservices, such as Numpy and Pandas Python libraries. Once the customized Docker images have been created, the scripts necessary for carrying out the tests have been created and the language used remains Python. Basically, the scripts dedicated to a microservice include the creation of the OpenWhisk action, the invocation of the same and, if the test requires it, the destruction of the corresponding container.

Face recognition

Face Recognition is the first microservice used for testing. The input corresponds to two images, each containing a person as the main subject, saved locally in jpg format. The two images are compared obtaining as output the correspondence or not of the two faces. To this purpose, the script is based on the following steps: (i) finding faces in the image, (ii) pose variations recognition, (iii) face coding, (iv) comparison of results. For each step, a different machine learning algorithm is used based on OpenFace [23] and dlib [24]. Once the values of these measurements are available, the service can answer to the question whether the images represent the same person.

Image conversion

Convert Images is another microservice used. This converts images from png to jpg format. The script created is based on an API, namely aspose.words [25], specialized in processing documents in various formats.

In order to guarantee a fair comparison of the two considered technological solutions, in our experiments we use the same sequence of images to test both the FaaS and the CaaS implementation of the microservice.

C. Test execution

The tests are carried out on two virtual machines hosted on a server provided by the University of Modena and Reggio Emilia. It is worth to note that the two VMs have the same characteristics: 4 threads and 4GB of RAM, and the same software and the same Operating System have been installed. In addition, OpenWhisk was installed on the FaaS-based system, in line with the tests to be performed.

In our experiments we consider two service invocation models, namely *sequential* and *interval*. In the first case, 3000 requests for each microservice are sent to the server sequentially and without concurrency, meaning that each request is sent when the answer to the previous request is received. In the second case, 1000 requests are sent for each microservice, with a time interval of 60 seconds that separates consecutive requests. The two execution methods aim to outline the differences between invocation of services involving a cold and warm start, that is the focus of our research.

For OpenWhisk, the sequential executions invokes the action, while for Docker executes the service inside the container. In both cases, we consider the use of just one container: given the sequential and not concurrent nature of the requests, this is sufficient to run the microservice. In the FaaS case, the sequential invocation ensures that OpenWhisk does not switch off the container corresponding to the action. Hence, it is not necessary for the container to be started at each invocation of the action because the container is always in a warm state and is ready to serve the request.

In the case of interval execution, requests are made one per minute, with the aim of measuring resources and response times when OpenWhisk shuts down the container. In the case of Docker the container is never switched off and is ready to serve the request, while in OpenWhisk the container is explicitly switched off (to avoid waiting for the container timeout) and is restarted for every request. This is used to test the microservices in the situation where the containers on OpenWhisk are in a cold state and for which a cold time is required due to the start of the container itself. The types of executions described above are designed to measure the effectiveness of the OpenWhisk system in both situations it can present, such as cold and warm, and to compare it with that of a system based exclusively on Docker.

III. Experimental Results

As already mentioned, the microservices runs on two systems with the same settings, one representative of Docker and the other OpenWhisk; requests for the microservices are sent by considering two scenarios: sequential and interval execution.

The main performance metrics considered in our analysis are the response times of the two microservices according to the two execution modes and the CPU and the RAM memory utilization.

We start our analysis focusing on the response time experienced by client accessing the two considered microservices. Fig. 1 and 2 show an histogram of the sampled response time

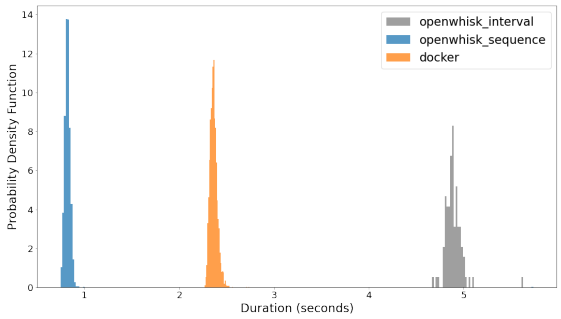

Fig. 1. Response times - Face recognition microservice

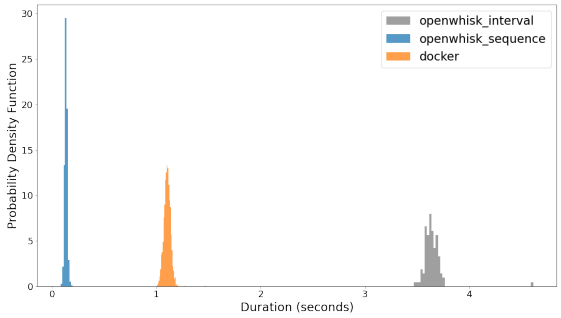

Fig. 2. Response times - Image conversion microservice

for face recognition and image conversion, respectively. Such representation is an approximation of the probability density function for the stochastic process of the request service time. The figure shows on the X-axis the value of the response time and on the Y-axis the probability density based on the measured samples of response time. In this experiment we compare the Docker case with the OpenWhisk technology in the cases of cold start and warm start. Differently from OpenWhisk, for the Docker case (orange histogram), there is no difference between cold and warm start, hence in the figures we see only one histogram, with an average response time in the order of 2.3 sec for the face recognition service and of 1.1 sec for the image conversion. On the other hand, for the FaaS approach we have a major performance impact on the response time given by the activation process. Focusing on the cold start (grey histogram) we observe a response time on average more than double compared to the docker case (4.8 sec and 3.6 for the two services). On the other hand, the warm start provides a significant performance benefit, with the FaaS solution outperforming the docker approach with a response time more than 4 times lower (0.7 sec and 0.2 sec for the two services).

The performance difference between the cold and warm start is a consequence of the container startup phase of OpenWhisk that requires a time on the order of seconds. The performance gain of OpenWhisk compared to docker when warm start occurs can be explained by the action pre-loading policy of the serverless technology that can carry out some preliminary phases of the service execution outside the request cycle.

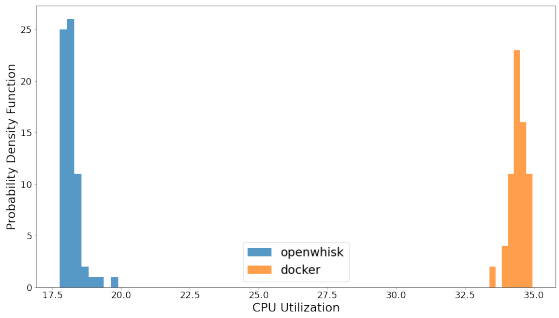

Fig. 3. CPU - Face recognition sequential execution

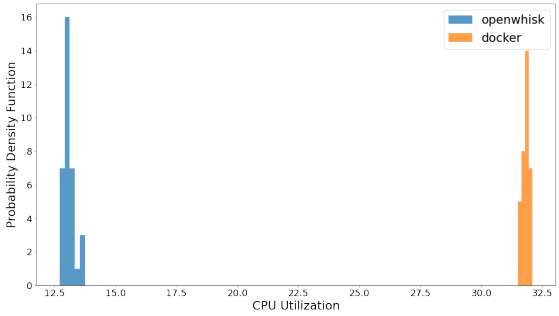

Fig. 4. CPU - Image conversion sequential execution

The histograms in Fig. 3 and 4 represent the CPU metric in sequential mode, respectively for face recognition and image conversion. The analyzed CPU has 4 cores. As for the previous image, the figure represents a probability histogram with the probability density on the Y-axis and the CPU utilization on the X-axis. Taking into account that in the sequential mode the requests are very frequent, it is possible to observe from the figures that both services executed on a docker-based system involve an average CPU utilization of about 20% compared to a system based on OpenWhisk. In particular, for face recognition the utilization is 18% for OpenWhisk and 34% for docker. For image conversion the utilization is 13% for OpenWhisk and 32% for docker. The values listed above show, in the sequential case, that the CPU consumption is always greater in docker than the consumption of OpenWhisk. Observing the trends of the CPU it is possible to draw some considerations: on docker the microservices are invoked every time starting from `docker exec` commands and in which the script to execute is indicated; on OpenWhisk, the action is initially created by associating it with the container created

ad hoc, which is the same used on docker, and at each execution of the script the command `wsk action invoke` is executed, specifying the name of the action. The reason is related to the nature of OpenWhisk: analyzing the structure of OpenWhisk, starting from the official documentation [26], it is observed that the actions are created and are kept in RAM memory, as we will see in the next analysis, ready to be invoked. This can save time time in invoking the action but only in starting, if necessary, the container in which it is executed. To support this, reference can be made to the response times described at the beginning of this chapter. For the same reason, even the CPU, in OpenWhisk, is particularly lightened by not having to interpret the script every time, as happens for docker.

We omit the results of the interval runs as the measured values are not meaningful due to the long idle times between subsequent invocations.

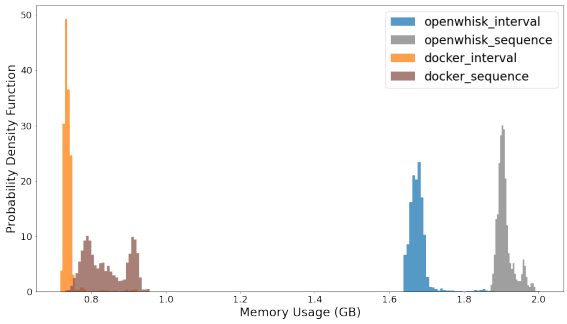

Fig. 5. RAM - Face recognition execution

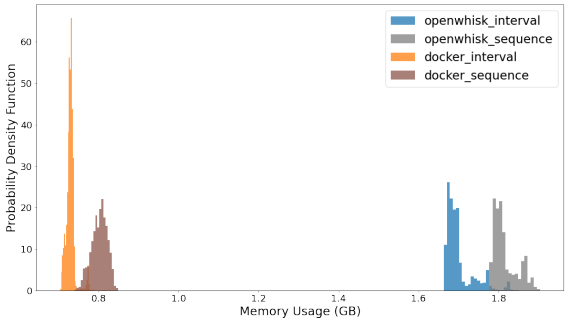

Fig. 6. RAM - Image conversion execution

Fig. 5 shows the histograms of the RAM memory usage measured for the face recognition service in the sequential and interval execution mode. Fig. 6, on the other hand, shows the histograms of the measurements made with the image conversion service, always in sequential and interval mode. Again the representation of the metrics is based on a probability histogram showing the probability density on the Y-axis and the RAM usage on the X-axis. In all figures the memory usage of the two systems is very different, highlighting that a docker system uses about half of the RAM used by an OpenWhisk system. For the face recognition service, we can observe two different situations: in the sequential execution mode, Docker uses about 900MB, while OpenWhisk usage is around 1.9GB; in the interval mode, the usage for Docker is 700MB, while for OpenWhisk it is about 1.7GB. With the image conversion service, in sequential mode, you have a usage of 800MB, while OpenWhisk of 1.8GB. For the interval mode, on the other hand, the situation is identical to that of face recognition, motivated by the fact that the system is not solicited as in the sequential mode. The trend described in the analysis section concerning the CPU must be taken into account and taken into consideration also for the use of RAM memory. In fact, as already mentioned, it is the nature of OpenWhisk that allows it to be more efficient both in terms of response times and in terms of CPU usage. This, however, negatively affects the use of RAM memory which is always more than double in OpenWhisk compared to docker. This RAM consumption is due to the internal components of OpenWhisk necessary to be always active in case of invocations of actions. Docker, on the other hand, always keeps the containers active, but every time it has to interpret and execute the scripts, as if they had never been executed before and this results in a higher CPU usage than OpenWhisk.

IV. Conclusions and Future Work

In this paper we proposed a comparison of Function-as-a-Service (FaaS) and Container-as-a-Service (CaaS) approaches to the deployment of microservices. Through the proposal of two different services, deployed using the two approaches we evaluated the response time and the resource consumption (i.e., CPU and memory). In our experiments we consider two operating conditions for the micro-services: sequence of requests and requests with a long inter-arrival interval, corresponding to warm- and cold-start of micro-services.

Our experimental evaluation demonstrates that, in the case of cold-start, the overhead of container initialization makes the use of FaaS significantly slower compared to the container-based approach. On the other hand, in the case of significant amount of sequential requests the FaaS approach seems significantly better performing. Another interesting finding is that, from a resource point of view, the operation of the FaaS approach can reduce the CPU utilization in the case of sequence of requests, but at the expenses of a significantly higher memory utilization.

In our future work, we plan to compare the performance of the FaaS and CaaS with a wider range of microservices characterized by different requirements to better understand the behaviour of the two considered model in terms of resource usage. Moreover, we aim to explore the performance of the FaaS model over server with different computational and memory resources, to understand the suitability of such paradigm in a edge computing architecture.

REFERENCES

[1] J. Nupponen and D. Taibi, "Serverless: What it is, what to do and what not to do," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 49–50.

[2] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *Journal of Systems and Software*, vol. 149, pp. 340–359, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121218302735

[3] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2021.

[4] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, p. 76–84, apr 2021. [Online]. Available: https://doi.org/10.1145/3406011

[5] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 159–169.

[6] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 181–188.

[7] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020, pp. 1–7.

[8] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'19. USA: USENIX Association, 2019, p. 21.

[9] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions," *Future Generation Computer Systems*, vol. 110, pp. 502–514, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X1730047X

[10] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, p. 110708, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301527

[11] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless MapReduce on AWS Lambda," *Future Generation Computer Systems*, vol. 97, pp. 259–274, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18325172

[12] "Cloud Functions - Serverless Microservices — Google Cloud Platform," 2022, – https://cloud.google.com/functions/.

[13] "Microsoft Azure Functions documentation," 2022, – https://docs.microsoft.com/en-us/azure/azure-functions/.

[14] "AWS Lambda - Serverless Compute," 2022, – https://aws.amazon.com/lambda/.

[15] AWS. (2022) Managing Lambda provisioned concurrency. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html

[16] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 64–78. [Online]. Available: https://doi.org/10.1145/3464298.3476133

[17] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2021, pp. 386–400.

[18] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018*, 2020, pp. 57–69.

[19] "Apache Openwhisk. Open Source Serverless Cloud Platform," 2022, – https://openwhisk.apache.org/.

[20] "Docker. Empowering app development for developers," 2022, – http://www.docker.com.

[21] C. Canali and R. Lancellotti, "A Fog Computing Service Placement for Smart Cities based on Genetic Algorithms," in *Proc. of International Conference on Cloud Computing and Services Science (CLOSER 2019)*, Heraklion, Greece, May 2019.

[22] R. Beraldi, C. Canali, R. Lancellotti, and G. Proietti Mattia, "Randomized load balancing under loosely correlated state information in fog computing," in *Proceedings of the 23rd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, ser. MSWiM'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 123–127. [Online]. Available: https://doi.org/10.1145/3416010.3423244

[23] "OpenFace," 2022, – https://cmusatyalab.github.io/openface/.

[24] "dlib C++ Library," 2022, – http://dlib.net/.

[25] "Aspose.Words for Python," 2022, – https://docs.aspose.com/words/python-net/.

[26] "Documentation OpenWhisk," 2022, – https://openwhisk.apache.org/documentation.html.