Performance Experiences From Running An E-health Inference Process As FaaS Across Diverse Clusters

George Kousiouris
gkousiou@hua.gr
Harokopio University
Athens, Greece

Aristodemos Pnevmatikakis
apnevmatikakis@innovationsprint.eu
Innovation Sprint
Brussels, Belgium

ABSTRACT

In this paper we report our experiences from the migration of an AI model inference process, used in the context of an E-health platform to the Function as a Service model. To that direction, a performance analysis is applied, across three available Cloud or Edge FaaS clusters based on the open source Apache Openwhisk FaaS platform. The aim is to highlight differences in performance based on the characteristics of each cluster, the request rates and the parameters of Openwhisk. The conclusions can be applied for understanding the expected behavior of the inference function in each of these clusters as well as the effect of the Openwhisk execution model. Key observations and findings are reported on aspects such as function execution duration, function sizing, wait time in the system, network latency and concurrent container overheads for different load rates. These can be used to detect in a black box manner capabilities of unknown clusters, guide or fine-tune performance models as well as private cloud FaaS deployment setup.

CCS CONCEPTS

• **General and reference** → **Measurement**; **Performance**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

Function as a Service, cloud, edge, performance analysis,benchmarking, Model inference

ACM Reference Format:
George Kousiouris and Aristodemos Pnevmatikakis. 2023. Performance Experiences From Running An E-health Inference Process As FaaS Across Diverse Clusters. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3578245.3585023

1 INTRODUCTION

The Function as a Service (FaaS) paradigm promises to deliver cloud native capabilities out of the box such as queue-based load levelling with competing consumers[7], easier packaging, deployment, management and scaled execution for function-wrapped application features. Cost models are more flexible, basing the cost on the number of invocations, the duration of a function execution as well as the memory size of the needed container[3]. However the process of migrating existing functionalities to the new model or setting up relevant private FaaS clusters is not without challenges, pitfalls and hidden parameters that may affect the application and system performance[21].

Combinatorial cloud/edge solutions have started to emerge, aiming at exploiting the serverless capabilities across the continuum[24] as well as the space-time relationship when distributing loads[10]. FaaS succeeds in creating abstractions of the resources, however this comes at the cost of having more obscure and hidden performance issues. So questions arise having to do with the setup and operation of these clusters and the target function.

The current work aims to highlight aspects of FaaS execution through benchmarking the execution of an adapted ML inference function across 3 different testbeds (in terms of hardware and Openwhisk parameters setup). The testbeds are scattered across different geographic locations (Greece, Netherlands and Sweden). In each case, the open source Openwhisk FaaS solution has been installed inside virtual machines. The goals of the work include:

- Quantify the effect of running the specific load across different clusters, from a hardware or configuration point of view and how this relates to the experienced performance, as well as get indications about the FaaS platform operation on each cluster setup (e.g. concurrent containers overhead, its effect on the cost model, cluster design and sizing choices etc.).
- Document experiences from practical issues of the migration and operation, such as function sizing, and observe the effect of a cluster choice on various delays (response time, latency, wait and service time of the function)

The paper proceeds as follows. In Section 2 related work is portrayed regarding FaaS platforms, performance investigation as well as E-health use cases. Section 3 includes details of the system, including the migrated function, the used testbeds and the load generation and measurement process. Section 4 analyzes the results and extracts main findings and observations, while Section 5 concludes the paper.

2 RELATED WORK

Openwhisk[9] is an open source FaaS platform that is also the backbone of the commercial IBM Cloud Functions offering. Openwhisk features include pre-warm containers, reuse of warm containers, many baseline function runtimes as well as the ability to use any custom docker image as a runtime, providing that the Openwhisk interface for activating the function is followed. Together with OpenFaaS[11], they are the two most widespread open source offerings for FaaS[1].

Performance modelling of serverless applications and platforms has gained significant attention recently. In [20], an analytical model

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0072-9/23/04.
https://doi.org/10.1145/3578245.3585023

is applied in order to investigate and predict aspects such as response time. Autoscaling mechanisms through reinforcement learning are investigated in [27] while scheduling and placement of application functions across the cloud/edge is investigated in [26] as a multi-objective optimization problem. What is common in all these cases is that data are needed for adapting or configuring the algorithms as well as knowledge of a number of baseline metrics like the function duration. The data collected in this paper can help fine-tune such models or be used as training data to extend model parameters with cluster sizes or configuration details. Cold starts, one of the main issues of serverless platforms, is not taken under consideration in our work. This is due to the fact that we anticipate such issues to be covered by the aforementioned higher level models, once other factor influences are known such as cluster sizes, function duration etc. Thus we focus on the latter.

Regarding the selected use case, the E-health domain has started to concentrate on the serverless paradigm in a variety of applications[18], stemming from data collection, interoperability functionalities, analytics and data storage and up to remote sensing, model training and model serving for patient monitoring. Created models may include prediction of a patient condition through inference[19], patient phenotyping[22] (the process of matching vectors of data from patients to the models of different phenotypes, returning the one each vector is more likely generated from) and data synthesis[25] for the creation of realistic synthetic data. Relevant architectures have also been determined for cases including cloud/fog/edge combinations of resources[13].

3 SYSTEM DESCRIPTION

3.1 Openwhisk FaaS Platform Overview

The Openwhisk Platform architecture (Fig. 1) follows a typical queue based load levelling pattern with competing consumers[2]. The API Gateway receives and validates the function invocation requests, which are then added in a Kafka queue.

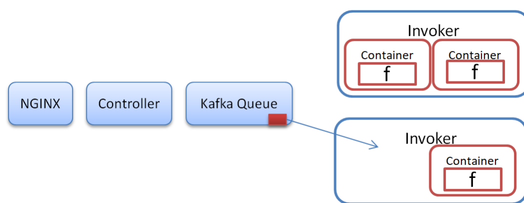

Figure 1: Overview of Openwhisk Architecture

Invokers running in each of the worker nodes consume the messages from the queue, raising one container to serve each function, if there is available memory. If not, the requests wait in the queue until a slot is freed. When a function execution finishes, the container that executed it remains available for a period of time (typically around 10 minutes). If a new invocation for the same function arrives within this period, it can reuse that container. More functions can be cramped inside one container if the system concurrency value is set accordingly. In our case we used the default value of 1 function per container.

3.2 Target Inference Function Structure

The target function consists of a model inference process[19]. This uses a pre-trained ML model in order to infer on the condition of a patient. Related data input indicates the patient's measurements across a vector of vital signs. These are fed as inputs to the model that classifies patient outlook as expected to be worsening, constant or improving. One function input can have more than one data rows for needed predictions. In our case, 10 rows were used for each invocation.

The overall stack of the used software for a function invocation appears in Fig. 2 . The function is created through the process defined in [16]. The Python script and the pre-trained model are included in the image during build time, through a customized Dockerfile. Although Openwhisk has a Python baseline template image to be used, this does not include any necessary dependencies. The latter can be included with the source code but they can not typically exceed a maximum file size that is rather small for AI/ML use cases. Hence the typical solution is to use custom Docker images[23] and use the according Openwhisk process[4].

The main function wrapper and runtime management layer is a Node-RED[1] flow (Fig.3). It undertakes the main interaction with the Openwhisk platform through serving and exposing the interface related REST methods for function invocation. Concentrating on the /run endpoint definition, first the received input JSON is converted into the command line arguments for the Python script in the Prepare CLA node. Then, the Python script is executed in the exec node. It uses NumPy version 1.23.4 (for array processing), Joblib v.1.2.0 (for reading stored models), Pandas v.1.5.2 (for data manipulation) and Tensorflow v.2.11.0 (for the ML model). The standard output and error streams are collected , joined and processed into the output by the Prepare response node. Error handling nodes are also present, to timely stop execution if something is wrong.

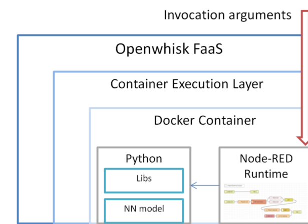

Figure 2: Software Stacks of the Function Execution

3.3 Cluster Description for Experimentation

The overall testbed setup appears in Fig. 4. The edge (located at Harokopio University-HUA) is considered a small installation (1 medium size node), close to the client, that due to its limited size is using the Openwhisk standalone version. The latter uses the local Docker Engine in order to execute the function containers. The function image is available locally, acting as a data caching layer (no need to fetch it). The other two are located on AWS (Sweden) and Azure (Netherlands). Only vanila VMs are used from these

[1] https://nodered.org/

providers. The AWS case uses OKD with Kubernetes 1.24.6 and m5.xlarge worker VM while the Azure one uses Kubernetes (v1.22.6) on a Standard D8ds v4 VM. Both installations use Openwhisk 1.0.0 while there is a private registry from which the images are drawn, if not available on the node. These clusters vary in terms of the type of worker nodes, as well as the memory allowed to be used by Openwhisk (Container Pool Memory) as detailed in Fig. 4. All the load generation is performed from the edge side (different resource than the Openwhisk node), given that we consider this the primary client app location. Thus setups include different hardware as well as container management layers, diverse locations as well as different Openwhisk settings. Further investigations may be performed in the future with more nodes and more target functions, like the two extra E-health functions mentioned in Section 2 (patient phenotyping and synthetic data creation).

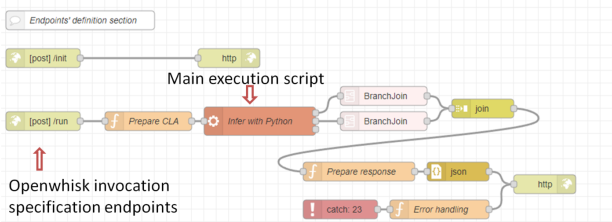

Figure 3: Node-RED flow inside the E-health function

The importance of the Container Pool Memory setting needs to be highlighted. This regulates the maximum allowed memory that Openwhisk is allowed to use, even if there is more physical memory available. Thus a cluster has a max number of available concurrent container slots equal to ContainerPoolMemory/M, for any given function with needed memory M. Any invocation that does not find a slot (free slot or finished warm container) is queued until one is available. This is depicted in the function's wait time statistic.

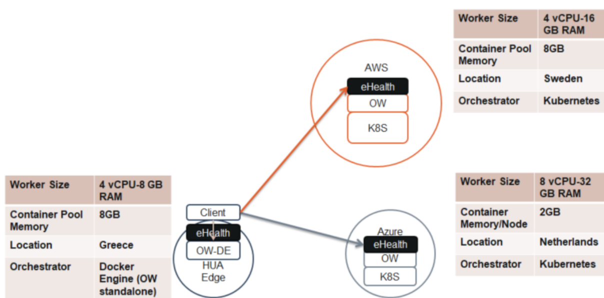

Figure 4: Overview of the 3 testbed clusters

3.4 Load Generator and Collected Metrics

The Load Generator used can be found in [6] and is built specifically for interacting with asynchronous APIs like the ones in Openwhisk. It is a set rate load generator, meaning it creates requests based on a defined uniform rate without blocking while waiting for the responses like other popular clients (e.g. Apache Jmeter). It further collects timestamps across the process as well as Openwhisk statistics in order to measure key intervals of the invocation. The complete list of measured points appears in Fig. 5.

The client load generation resides at the edge layer. This is done based on the assumption that this is the main operational hub of the client. The client runs as a separate Node-RED flow in another server at the edge, in order not to affect the main edge node processing. The load generator performs the individual homogeneous calls for the set duration and rate of the configuration. Then it calculates the average and the standard deviation, as well as stores the raw data for future use. It further validates the achieved average rate of messages. The load generator is also available as a docker image[5].

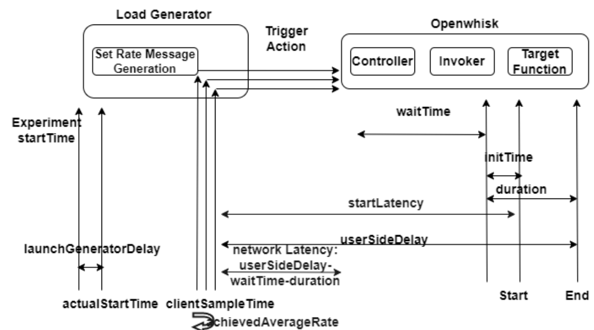

Figure 5: Load Generator and Measurement Points Collected

4 EXPERIMENTATION RESULTS

4.1 Measurement Series Implementation

Three distinct function invocation rates were used (12, 30 and 60 messages per minute) for 3 distinct function memory sizes (256, 512,1024MB) on each testbed. Before running the experiments, the clocks across the testbeds were NTP-synchronized using the chrony Ubuntu package. This is necessary since we are collecting timestamps from different locations and comparing them to extract metrics, as detailed in Section III.

A prewarming run was performed in each case in order to remove contamination of the results from the initial cold starts. This was done since it seems that in Openwhisk the container initialization time is included in the function execution duration, although it is also reported separately. After the prewarm run and while the warm containers were available, data were collected during the main run for a duration of 600 seconds for each case. Overall 4890 samples were collected. The data have been made available publicly[15].

4.2 Function Execution Duration

In Fig. 6 the results are portrayed for the pure service time (i.e. pure function execution duration in the Openwhisk terminology, without the waiting time).

4.2.1 Function Memory Sizing. By comparing the three different memory sizes in the HUA low rate case (thus where no concurrency

effects from many parallel running containers affect the results), it can be observed that the increased memory allocation for this function does not provide any significant performance advantage by itself. The average value (6402 mseconds) in the 256MB allocation is very close to the averages of 6370 and 6251 in the 512 and 1024MB allocations accordingly. What can be observed is the lack of values for the AWS and Azure cases for the 256MB allocation.

In fact in this case the function kept failing to execute, with no apparent reason and although the exact same version was executed in the HUA testbed. After observing the function memory consumption, it was determined that there was a spike, raising it momentarily to around 400MB. At that time the Kubernetes environment immediately takes action. In fact it does not kill the pod itself, which would be an immediately observed action, but the Python process inside the pod that generates the high memory consumption. This leads to unexpected function behaviour (stalling of the flow) and difficult to debug cases (function timeout as an outcome of the flow stalling). On the other hand, the standalone Openwhisk installation (HUA edge) uses the local Docker Engine not enforcing such strong measures, thus enabling a successful execution.

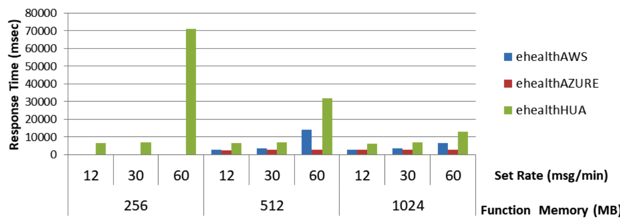

Figure 6: Average Function Execution Duration (service time)

Finding 1: In the specific function, increased memory does not help performance. The optimal value from a cost-benefit point of view is 256MB however for functional reasons (temporary memory spikes) we need to use the more costly 512MB option as a base for a K8S environment.

Observation 1: If Kubernetes was more tolerant in the way it treats temporary memory spikes, we could use the 256MB cost saving option and also avoid difficult to detect errors. Killing directly a process for even a momentary violation seems too strict. A time window of tolerance could be useful in such cases. This could also aid the FaaS provider to have more container slots for the same amount of memory.

4.2.2 Concurrent containers effect. One interesting conclusion can be extracted from the HUA 60 rate series for the three memory values. A significant deterioration in function execution duration can be seen in the 256MB case (70000 milliseconds of execution) compared to the 512MB case (31671 mseconds) and 1024MB (13027 mseconds). This can be attributed to the fact that in the 256MB case, the maximum concurrent containers are 8GB/256MB=32. Given that it is a high rate scenario, almost all of these slots will be used concurrently, leading to reduced CPU time assigned per container as well as interference effects on cache misses etc. In the 512 case the max containers are 16, whereas in the 1024 case they are 8. In the low rate case, where only 1-2 containers are used concurrently, the relevant value is 6402 mseconds. The relevant graph appears in Fig. 7. A relevant deterioration can also be seen in the AWS case, in which the 30-512 scenario that has approximately 5 concurrent containers shows an increase to 3492 milliseconds (from the 2787 milliseconds of the 12-512 case that has 1 active container) and the 60-512 scenario that has 16 (8GB/512) active containers shows an increase to 14229 mseconds. On the other hand, the Azure case that is limited to 4 (2GB/512) and 2 (2GB/1GB) slots in these scenarios does not show any significant increase in the function duration. Another aspect of concurrency is a number of failures ($\tilde{5}$%) in the AWS case. This was observed to be from image fetching errors from the private registry used, which acted as a bottleneck in some limited simultaneous cold-start requests. No errors were observed in the HUA case, since the image was included in the node, acting as a data caching layer. Azure did not demonstrate this due to the lower container slots available.

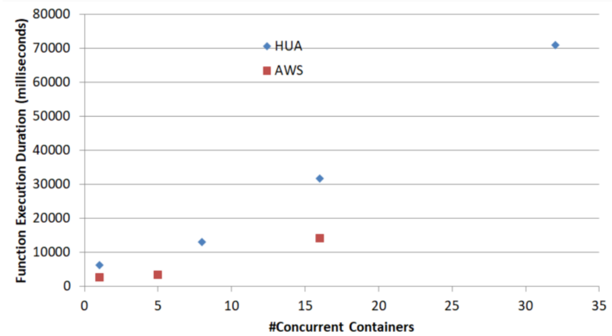

Figure 7: Avg Function Execution Duration Compared to Number of Concurrent Containers in the Edge and AWS

Finding 2: The performance degradation from the concurrent containers in the node can have extreme effects. This could be reduced from one point of view (CPU sharing time) if strict scheduling strategies (like real time scheduling) and CPU quotas are used. However even in this case, studies[17] have shown that still degradation can be extremely significant. For reducing the back-end stress and user costs, dynamic batching approaches ([8],[14]) have also proven to be very effective.

Observation 2: Too many concurrent containers in the cluster, potentially from different customers, result in extended durations for a given, deterministic client function with the same input. However in the FaaS model the cost depends on the function duration. This poses the question: Why does the client have to pay more for a worse quality of service, because the provider has more clients?

Observation 3: The concurrency overhead affecting function duration means that the system's service rate (rate of finishing functions) is affected by the number of clients in the system. This is especially true in small scale and highly loaded systems. This needs to be considered when considering queuing models or other techniques that imply that the service rate is not state dependent.

4.2.3 *Best Performing Baseline Instance.*

For detecting the capabilities of the main VM instance used, we can check the measurements of the low rate case across the three clusters and especially the function duration. This does not include any wait or container initialization time, thus only the pure capability of each machine on the given problem can be observed. From this, in the 512MB case Azure demonstrates the best performance (2477 msecs) while AWS comes in second (2787 msecs) and the Edge HUA node third (6370 msecs). Similar results appear for the 1024 case. From an examination of the instance type descriptions, both cloud types are based on the Intel Xeon Platinum 8000 series, although Azure on the 3rd[2] and AWS on the 1st or 2nd generation[3]. The edge node is based on the less powerful Intel Xeon E312xx Sandy Bridge.

Finding 3: From the measurements we can conclude that the best performing case is in the Azure cluster while AWS comes in second with a small difference. The main benefit in this case is the ability to detect this difference from a macroscopic point of view. A more detailed analysis could also include the cost of every instance in order to observe the relative cost-benefit ratio like in the case of [12]. Also the endurance to concurrent containers would be interesting to check, although in the specific case it was not feasible due to the difference in the Container Pool Memory setting.

4.3 Waiting time investigation

The wait time is reported by Openwhisk and retrieved by the Load Generator. The average in each scenario has been plotted in Fig. 8. From this, the effect of the parameter of the Container Pool Memory can be observed. Although the Azure VM is a larger one (32GB of RAM and 8 cores), the usage of a low value (2GB) for this parameter limits the available container slots. Thus higher queuing times are observed in the high rate (60msg/min) compared to the AWS case, even though the latter is smaller (4 cores and 16GB of RAM). The use of 8GB of Container Pool Memory helps the AWS testbed to serve more concurrent requests. The Azure cluster starts having wait delay from the medium rate.

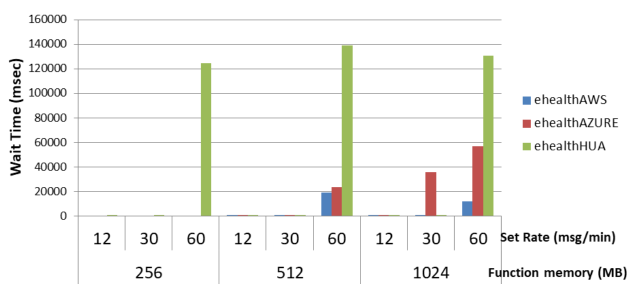

Figure 8: Average Wait Time for Function Invocation

Finding 4: Careful consideration needs to be applied when deploying a FaaS cluster. The number of setup parameters that might affect performance is significant and can have a large effect on the cluster.

Observation 4: Interesting trade-offs can be investigated in this case if we combine it with the concurrent containers analysis. Higher numbers of available container slots reduce wait time but increase execution duration due to concurrency. This is reminding of the context switching problem in typical web servers where thread limits need to be set. For cluster sizing, more nodes with less memory per node could help reduce wait time while not skyrocketing interference effects. From an energy point of view, if the total response time is the same but in one case it consists more of wait time than execution, this would aid the specific execution to use up less energy.

4.4 Networking aspects of cloud-edge trade-off

The network aspect, as shown in Fig. 5, includes some of the initial latencies in the incoming Openwhisk layers like the Nginx reception. The pure network latency could be easily retrieved between the network endpoints but it would then not depict these entry layers. The data are included in Fig. 9

Examined latencies follow a logical result, indicating higher numbers for more remote geographic locations. The client as mentioned in Fig 4 is located in Greece, hence the client to edge latency is minimal. AWS has the largest distance (Sweden to Greece) and portrays the highest latency, whereas Azure (Netherlands to Greece) is in the middle.

Finding 5: In the particular case, the network latency is rather small (approximately 8%) compared to the function duration. So the latter is not considered a key selection factor. However in lower function duration cases it could represent a significant percentage of the total delay.

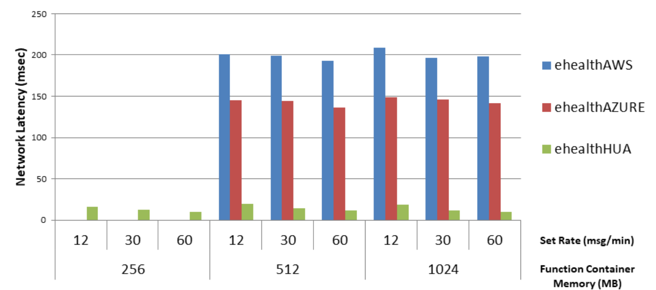

Figure 9: Average Estimated Network Latency from subtracting duration and wait time from total user side delay

4.5 Total User Side Delay

The total user side delay (Fig. 10) is measured as the response times of the function invocations from the client generator. As such it includes all intermediate times. From that it can be observed that for the middle memory scenario, the 4 available container slots (2GB Container Memory/512MB function memory=4 slots) in the Azure testbed are sufficient in order to keep wait time relatively short, even in the high rate case. Thus the benefit of the other factors (reduced latency and execution time) prevails. In the high function memory scenario, the slots are half due to the doubling of

[2]https://learn.microsoft.com/en-us/azure/virtual-machines/dv4-dsv4-series
[3]https://aws.amazon.com/ec2/instance-types/m5/

the function memory (1GB), and now the wait time is considerable in the medium and high rate scenarios (as was evident from Fig. 8 as well). Thus in this case the execution on AWS is better.

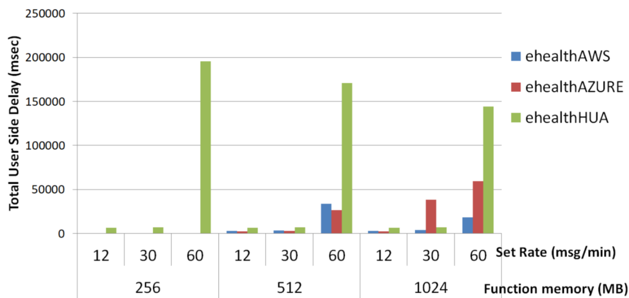

Figure 10: Average Total User Side Response Time

Observation 5: Knowing the performance of each cluster across different scenarios and mapping this to the total response time can enable a more dynamic application operation through e.g. a routing decision maker, forwarding requests to the 3 locations depending on the current conditions of execution or based on functions with different priorities.

5 CONCLUSIONS

As a conclusion, the behaviour of the examined function changes significantly based on different cluster characteristics and incoming traffic. The analysis led to quantified findings and evaluation of each setup.

The process was able to accurately detect cluster characteristics macroscopically. Best performing nodes, immediate availability in high rate scenarios or interference resulting to deviant QoS can be highlighted. Thus it can aid in cases where no information on the internals of a cluster is available. Furthermore, the observations and findings can aid teams provisioning private FaaS clusters as to how to configure the resources needed for such a cluster. As an example, larger number of nodes with smaller used memory per node can achieve the same available function execution slots while minimizing concurrency effects.

For the future, extensions may include a more accurate prediction of the concurrent containers effect and how this changes with mixed function workloads, especially different types and sizes of E-health functions as well as more nodes per cluster. The inclusion of cost in the performance statistics can lead to scores per cluster type. Finally, the inclusion of energy consumption can further optimize the runtime operation, trading larger execution durations (due to concurrency effects) for idle wait time in high multitenancy scenarios, through the regulation of the available container slots.

ACKNOWLEDGMENT

The research presented has received funding from the European Union's Project H2020 PHYSICS (GA 101017047).

REFERENCES

[1] 2018. Comparison of FaaS Open Source Platforms, available at:. https://winder.ai/a-comparison-of-serverless-frameworks-for-kubernetes-openfaas-openwhisk-fission-kubeless-and-more/.

[2] 2019. Openwhisk Documentation and Architecture. https://openwhisk.apache.org/documentation.html

[3] 2022. *Amazon Lambda Pricing Model, available at:*. https://aws.amazon.com/lambda/pricing/

[4] 2022. Openwhisk Docker Action documentation. https://github.com/apache/openwhisk/blob/master/docs/actions-docker.md.

[5] 2022. PHYSICS Load Generator Docker Image, available at:. https://hub.docker.com/r/gkousiou/physicspef_loadgenclient.

[6] 2022. PHYSICS Load Generator Subflow, available at:. https://flows.nodered.org/flow/53bf7addb6ef140ab7e9395c9a9feb1b/in/HXSkA2JJLcGA.

[7] 2022. Queue Based Load Levelling with Competing Consumers Pattern Design, available at:. https://learn.microsoft.com/en-us/azure/architecture/patterns/competing-consumers.

[8] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing Inference Serving on Serverless Platforms. *Proc. VLDB Endow.* 15, 10 (sep 2022), 2071–2084. https://doi.org/10.14778/3547305.3547313

[9] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. 2016. Cloud-Native, Event-Based Programming for Mobile Applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems* (Austin, Texas) *(MOBILESoft '16)*. Association for Computing Machinery, New York, NY, USA, 287–288. https://doi.org/10.1145/2897073.2897713

[10] Ryan Chard, Jim Pruyne, Kurt McKee, Josh Bryan, Brigitte Raumann, Rachana Ananthakrishnan, Kyle Chard, and Ian T. Foster. 2023. Globus automation services: Research process automation across the space-time continuum. *Future Generation Computer Systems* 142 (2023), 393–409. https://doi.org/10.1016/j.future.2023.01.010

[11] Alex Ellis et al. 2019. OpenFaaS: Serverless Functions, Made Simple. https://github.com/openfaas/faas

[12] Athanasia Evangelinou, Michele Ciavotta, Danilo Ardagna, Aliki Kopaneli, George Kousiouris, and Theodora Varvarigou. 2018. Enterprise applications cloud rightsizing through a joint benchmarking and optimization approach. *Future Generation Computer Systems* 78 (2018), 102–114. https://doi.org/10.1016/j.future.2016.11.002

[13] Masoumeh Hajvali, Sahar Adabi, Ali Rezaee, and Mehdi Hosseinzadeh. 2022. Software Architecture for IoT-Based Health-Care Systems with Cloud/Fog Service Model. *Cluster Computing* 25, 1 (feb 2022), 91–118. https://doi.org/10.1007/s10586-021-03375-4

[14] George Kousiouris. 2021. A self-adaptive batch request aggregation pattern for improving resource management, response time and costs in microservice and serverless environments. In *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. 1–10. https://doi.org/10.1109/IPCCC51483.2021.9679422

[15] G. Kousiouris. 2023. *Data Collection of HotCloudPerf Submission, available at:*. https://github.com/gkousiouris/measurements/tree/main/hotcloudperf2023

[16] George Kousiouris, Szymon Ambroziak, Domenico Costantino, Stylianos Tsarsitalidis, Evangelos Boutas, Alessandro Mamelli, and Teta Stamati. 2022. Combining Node-RED and Openwhisk for Pattern-based Development and Execution of Complex FaaS Workflows. https://doi.org/10.48550/ARXIV.2202.09683

[17] George Kousiouris, Tommaso Cucinotta, and Theodora Varvarigou. 2011. The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks. *Journal of Systems and Software* 84, 8 (2011), 1270–1291. https://doi.org/10.1016/j.jss.2011.04.013

[18] Anisha Kumari, Ranjan Kumar Behera, Bibhudatta Sahoo, and Sanjay Misra. 2022. Role of serverless computing in healthcare systems: Case studies. In *Computational Science and Its Applications–ICCSA 2022 Workshops: Malaga, Spain, July 4–7, 2022, Proceedings, Part IV*. Springer, 123–134.

[19] Sofoklis Kyriazakos, Aristodemos Pnevmatikakis, Alfredo Cesario, Konstantina Kostopoulou, Luca Boldrini, Vincenzo Valentini, and Giovanni Scambia. 2021. Discovering Composite Lifestyle Biomarkers With Artificial Intelligence From Clinical Studies to Enable Smart eHealth and Digital Therapeutic Services. *Frontiers in Digital Health* 3 (2021). https://doi.org/10.3389/fdgth.2021.648190

[20] Nima Mahmoudi and Hamzeh Khazaei. 2022. Performance Modeling of Metric-Based Serverless Computing Platforms. *IEEE Transactions on Cloud Computing* (2022), 1–1. https://doi.org/10.1109/TCC.2022.3169619

[21] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. 2022. A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions. *ACM Comput. Surv.* 54, 11s, Article 222 (sep 2022), 36 pages. https://doi.org/10.1145/3510412

[22] Harm op den Akker, Miriam Cabrita, and Aristodemos Pnevmatikakis. 2021. Digital Therapeutics: Virtual Coaching Powered by Artificial Intelligence on Real-World Data. *Frontiers in Computer Science* 3 (2021). https://doi.org/10.3389/fcomp.2021.750428

[23] Efterpi Paraskevoulakou and Dimosthenis Kyriazis. 2021. Leveraging the serverless paradigm for realizing machine learning pipelines across the edge-cloud

continuum. In *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. 110–117. https://doi.org/10.1109/ICIN51074.2021.9385525

[24] Panos Patros, Melanie Ooi, Victoria Huang, Michael Mayo, Chris Anderson, Stephen Burroughs, Matt Baughman, Osama Almurshed, Omer Rana, Ryan Chard, Kyle Chard, and Ian Foster. 2022. Rural AI: Serverless-Powered Federated Learning for Remote Applications. *IEEE Internet Computing* (2022), 1–5. https://doi.org/10.1109/MIC.2022.3202764

[25] Aristodemos Pnevmatikakis, Stathis Kanavos, George Matikas, Konstantina Kostopoulou, Alfredo Cesario, and Sofoklis Kyriazakos. 2021. Risk Assessment for Personalized Health Insurance Based on Real-World Data. *Risks* 9, 3 (2021). https://doi.org/10.3390/risks9030046

[26] Renchao Xie, Dier Gu, Qinqin Tang, Tao Huang, and Fei Richard Yu. 2022. Workflow Scheduling in Serverless Edge Computing for the Industrial Internet of Things: A Learning Approach. *IEEE Transactions on Industrial Informatics* (2022), 1–10. https://doi.org/10.1109/TII.2022.3217477

[27] Anastasios Zafeiropoulos, Eleni Fotopoulou, Nikos Filinis, and Symeon Papavassiliou. 2022. Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms. *Simulation Modelling Practice and Theory* 116 (2022), 102461. https://doi.org/10.1016/j.simpat.2021.102461