UNIVERSITY OF LEEDS

Proceedings Paper:
Djemame, K orcid.org/0000-0001-5811-5263, Parker, M and Datsev, D (2020) Open-source Serverless Architectures: an Evaluation of Apache OpenWhisk. In: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). CIFS - 2nd International Workshop on Cloud, IoT and Fog Systems (and Security), 07-10 Dec 2020, Leicester, UK. IEEE , pp. 329-335. ISBN 978-1-6654-1563-7

https://doi.org/10.1109/UCC48980.2020.00052

Open-source Serverless Architectures: an Evaluation of Apache OpenWhisk

Karim Djemame
School of Computing
University of Leeds
Leeds, UK
Email: K.Djemame@leeds.ac.uk

Matthew Parker
School of Computing
University of Leeds
Leeds, UK

Daniel Datsev
School of Computing
University of Leeds
Leeds, UK

***Abstract*—The serverless computing paradigm ushers in new concepts for running applications and services in the cloud. Currently, commercial solutions dominate the market, though open-source solutions do exist. As a consequence of this, there is little research detailing how well the different open-source solutions perform. In this paper, one such open-source solution, Apache OpenWhisk, is investigated to shed light on the capabilities and limitations inherent of such serverless computing architecture, and principally to provide further research on this particular solution's performance. This is accomplished through an extensive evaluation of OpenWhisk, involving a variety of experiments and benchmarks.**

***Keywords*-Serverless Architecture, Openwhisk, Cloud Computing, Containerisation, Performance Evaluation**

I. Introduction

Recent advancements in cloud computing and virtualisation have led to the emergence of serverless computing; a technology which leverages container-based virtualisation to deploy applications and services. The goal of serverless computing is to provide isolated environments that abstract underlying technologies and expose small runtime containers for users to run functions as code [1].

A serverless computing system is an ideal solution to build and optimise any Internet of Things (IoT) operation with zero infrastructure and maintenance costs and little-to-no operating expense as it allows IoT businesses to offload all of a server's typical operational backend responsibilities. Moreover, such system is a natural fit for edge computing applications as serverless computing also supports the protocols which IoT devices require in actual deployment conditions.

Serverless computing has seen widespread adoption from tech industry giants such as Amazon [2], Microsoft [3] and Google [4], as well as the public domain, with open-source projects like *Apache OpenWhisk* [5], *Fission* [6], *IronFunctions* [7] and more. It offers scalability, fault tolerance and cost benefits, but also comes with a set of drawbacks related to the execution environment that affects the viability and design of applications [8]. Moreover, there are a number of performance related challenges in serverless computing such as unreliability, large overheads and an absence of benchmarks [9]. Investigations into various aspects of serverless architectures are therefore required to guide the decision making process.

The lack of benchmarks and research in general, particularly within open-source serverless computing, is a key issue. Further research into the performance of open-source serverless architectures would provide greater insight into their capabilities and increase awareness of their potential as alternatives to commercial offerings. Given that serverless computing is still a relatively new technology, only a handful of open-source serverless architectures currently exist. One such framework, Apache OpenWhisk [5] is quickly becoming one of the most popular options, likely due to the exposure its received via IBM Cloud Functions, IBM's commercial serverless solution based on Apache OpenWhisk, and its large number of contributors. OpenWhisk is an ideal candidate for assessing the capabilities of serverless computing and is the chosen framework for this serverless architectures investigation.

The contributions of this paper are:

1) we propose a cloud-based technical solution for benchmarking and analysis of Apache OpenWhisk platform using a set of test functions;
2) we demonstrate OpenWhisk's performance in terms of effectiveness and efficiency.

This paper is structured as follows: section II briefly provides some background information on serverless computing and OpenWhisk's architecture. The related work is reviewed is section III. The experimental environment setup and the test functions for various serverless use cases are described in section IV. Section V presents the results of the experiments and compares OpenWhisk's performance against two other different solutions, *Docker* and *native*. Section VI concludes the paper and describes future work.

II. Background

The last ten years saw an evolution in cloud platform hosting: 1) from buying or renting physical servers to run applications and paying for those servers to be maintained; 2) to the widespread adoption of virtualisation, allowing one server to be treated as many software-defined Virtual Machines (VMs). Containerisation is seen as a refinement

and intersection of virtualisation and configuration management; 3) to Platform-as-a-Service (PaaS) as the next level of abstraction away from running own servers and deployment processes, and 4) to serverless which is similar to PaaS, but allows for small fragments of code to be deployed to support building self-scaling applications.

The term *serverless* doesn't imply that there are no servers involved; servers simply become transparent to users. It has been linked to mainly two service models, similar to the ones that originally emerged with the rise of cloud computing: 1) *Backend as a Service(BaaS)*: refers to services that offer features traditionally implemented by *back-end* applications such as databases or API servers. Users can incorporate them in their *front-end* web applications without the provisioning, setup and management of servers. Although similar to PaaS, they are more full-featured, implementing server-side logic such as user authentication or push/pull notifications which PaaS offerings forego in favour of more flexibility, and 2) *Function as a Service(FaaS)*: this model allows users to develop their application logic by compositing event-driven executable elements called *functions*. These functions are executed inside ephemeral containers, taking advantage of container virtualization to quickly provision resources for the duration of the execution. Most notably these containers are managed by the providers and scale automatically in number based on demand. FaaS is the most prominent model of serverless computing and has seen widespread adoption by both industry and open-source communities. Notable platforms include AWS Lambda [2], Microsoft Azure Functions [3] and Google Cloud Functions [4].

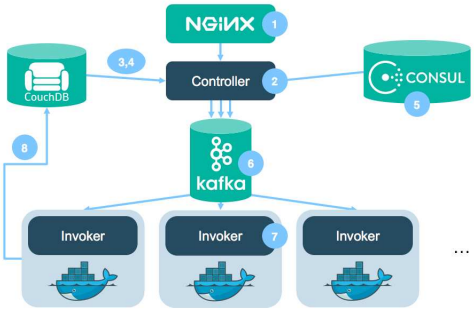

Figure 1. OpenWhisk Architecture. Source: [4]

OpenWhisk follows a simple event-driven architecture – functions are being triggered in re-sponse to events originating from direct invocations to the OpenWhisk API, or external servicesthat are integrated in the platform through specialised packages. Its architecture is shown in Figure 1. The numbers in the diagram show the flow in execution and deployment. In the first step, nginx [10] is used to expose HTTP endpoints to clients so functions can be requested. Once a request is issued, it hits the controller which performs authorisation and authentication of every request. Then, the controller interacts with a CouchDB [11] instance to verify credentials, namespaces and all things associated with the requested function. After this, the controller interacts with a Consul [12] instance for service discovery. The real-time data pipeline tool Kafka [13] is then used for communication between the controller and invokers. Invokers perform all the heavy lifting, handling container deployment and resource allocation to create a runtime for function execution.

OpenWhisk is developed as a platform built on top of Docker [14] and can therefore be deployed on a number of cloud and IoT infrastructures as well as container technologies.

III. Related Work

For a map of state-of-the-art research on the topic of FaaS platform and tooling engineering together with analysis of relations of the proposed concepts to existing solutions, the reader is referred to [15]: the mapping study on engineering FaaS platforms and tools provides insights on publication trends, the common challenges and drivers for research as well as information on industry participation in research publications. The reader interested in the evaluation of *production serverless computing environments* (AWS Lambda, Microsoft Azure, Google Cloud Functions and IBM Apache OpenWhisk) is referred to [16], [17], [18]. Next, two main areas of related research are reviewed: the first one is about the general design limitations of serverless platforms, in particular relating to runtime performance, and the second one focuses on performance evaluations of open-source serverless platforms, similar to the one investigated in this paper.

Design Limitations. There has been extensive research around factors affecting function execution performance. Manner et al. [19] consider the cold start problem and highlight the impact of the choice of language runtime; compiled languages lead to 2-3 times higher latency compared to interpreted languages like JavaScript. Their tests were performed on AWS and Azure and interestingly the measurements differ significantly between the platform and client side, further reinforcing the decision to measure both raw function execution time and API request latency.

The cold start problem has also been investigated by Lloyd et al. [20], where they note a 15× factor differential between warm and cold start times. They suggest providers keep containers alive for longer to offset the performance impact. This needs to be balanced with the cost of maintaining unused containers that the user is not charged for. A choice of language that minimizes cold start times is therefore important.

Baldini et al. [8] also acknowledge cold start as one of the main challenges of serverless platforms and additionally highlight the limitations of the current programming model, noting expressivity and composability as the main areas of improvement. Maas et. al [21] drill further and identify pain

points for current language runtimes in cloud deployments; they develop seven tenets for runtime design, proposing a library-based approach for integration with existing programming languages. Spillner [22] proposes *Snafu*, a FaaS runtime implementation focusing on flexibility and ease-of-deployment, with improved performance over AWS Lambda in a variety of configurations. This showcases the importance of runtime design; we can therefore expect serverless platform choice to affect performance, even for the same languages.

Open-Source Serverless Platforms. The evaluation of open-source serverless frameworks has been investigated by Mohanty [23] considering a performance evaluation of Kubernetes-based platforms (Fission, Kubeless and OpenFaaS) and focusing on feature comparisons and auto-scaling performance as the number of users and requests increases. Similar research by Li et al. [24] compare four platforms (Kubeless, OpenFaaS, Nuclio, Knative), noting the underwhelming state of the auto-scaling mechanisms which might prevent production deployments. This is another facet to consider when measuring response latency, as any decently-sized production environment will be faced with these issues. This is most likely one more reason why most companies turn to commercial offerings, and literature in this field is sparse.

Little attention has been given to OpenWhisk assessment with respect to performance. The work in [25] compared OpenWhisk against other serverless solutions, including commercial ones, whilst a performance comparison between different function implementations in OpenWhisk is found in [26].

To summarise, previous research involved benchmarking OpenWhisk against itself using different programming languages for function execution [26] and benchmarking OpenWhisk against other serverless solutions [25].

In essence, this paper contributes further to research in this field: 1) experiments are designed to assess OpenWhisk's capabilities through performance intensive tasks, and 2) experiments are performed to benchmark OpenWhisk in three performance areas against alternative methods which mimic function execution performed by serverless computing.

IV. Experimental Design

The plan is to evaluate OpenWhisk's performance using a single programming language for implementation of separate functions, with each function targeting a single hardware resource. Functions will need to be designed such that they are intensive for the resource they are targeting. Benchmarking these against alternate solutions to better measure OpenWhisk's performance is another plan for evaluation.

Cloud testbed. The experimentation was performed on a Cloud testbed available at the University of Leeds comprising a 14 node cluster. It uses Open Nebula 4.10.2 [27] and Zabbix 2.4.4 [28] for monitoring. The typical node that was considered for measurement is a Dell PowerEdge R430 Server commodity server with two 2.4GHz Intel Xeon E5-2630 v3 CPUs with 128GB of RAM, a 120GB SSD hard disk and an iDRAC Port Card.

Virtual Machine. Table I details the resources allocated to the Virtual Machine setup for OpenWhisk installation. The operating system used is Devuan; a fork of the Debian Linux distribution. This particular Linux distribution was chosen as the image was readily available in the OpenNebula marketplace and offered the most disk space, which was required mainly to support OpenWhisk's installation amongst other things. With OpenWhisk running persistently within the VM, it was then possible to begin prototyping the functions for OpenWhisk to invoke.

Table I
Virtual Machine Specification

CPU	vCPU	Memory	Operating System
16	16	16GB	Devuan GNU+Linux

Measurements and Metrics. The performance of an application is typically measured by taking into account a number of Key Performance Indicators (KPIs). Often, KPIs are split into two types, namely efficiency-oriented KPIs and service-oriented KPIs [29]. The former includes throughput and utilisation which determine an application's effectiveness at using the resources available to it, while the latter includes availability and runtime which measure an application's ability to provide a service to its end users. For the purpose of measuring the performance of Apache OpenWhisk, a combination of both KPI types are used.

Runtime is easily measured regardless of what an experiment entails. Alternatively, the tasks involved in each experiment decide which resources are utilised and to what extent. As such, it is reasonable to design each experiment in a way which targets a specific resource and exhausts it to varying degrees, the latter being achieved through the use of iterative methods. The resources chosen for experimentation are those most frequently found in performance evaluation.

CPU. The central processing unit (CPU) is perhaps the most popular resource to evaluate and benchmark for performance due to its sheer importance in computer systems. The most common method for generating consumption of the CPU is to create a program which performs a multitude of complex numerical calculations. Examples of this include calculating Pi and computing prime numbers. Between these examples, a range of iterative algorithms exist. For this experiment, calculation of Pi is the preferred choice for expending the CPU as most iterative algorithms for calculating Pi are easily implemented and do not depend on data structures. Hence, in theory, this experiment has little effect on other system resources and is therefore unlikely to be at risk of a bottleneck. The selected algorithm for calculating

Pi is the Bailey-Borwein-Plouffe (BBP) formula [30], and is chosen due to its simplistic design and ease of implementation, i.e. consideration of the number of iterations N and decimal places precision.

Memory. After the CPU, system memory is probably the next-most popular resource to analyse. In addition to this, it is the main determinant for pricing in commercial serverless solutions such as AWS Lambda [2]. There are few generally accepted methods for depleting memory and none which avoid impacting other resources. Thus, the method chosen for this experiment is performing the multiplication of two matrices of dimension N $\times$ N.

Network Although generally considered less important than CPU and memory, network resources are innately dependent on within distributed systems. In the case of serverless computing, prime examples of reliance on network resources are in supporting requests from users and responses from functions. As a result, much like the other resources, ove rheads affecting the network impacts overall performance and user experience.That said, the greater interest in CPU and memory experimentation means that this experiment is less of a focus and so simpler by design. The experiment itself purely involves HTTP GET requests to a locally hosted Web service.

Experiment Execution and Benchmarks. To execute the experiment tasks on OpenWhisk, a new OpenWhisk action is created for each of them. As listed in Table I, the VM in which OpenWhisk runs has 16 CPU cores. Since each action - or function - essentially runs on a distinct thread, and a single thread is limited to no more than 1 core, to achieve 100% CPU utilisation in the VM, 16 functions are run concurrently.

Each experiment task is executed using two other different solutions, *Docker* [14] and *native*, which can be considered more contemporary solutions for accomplishing the same tasks. The purpose of this is to provide fair benchmarks for OpenWhisk to contrast with. This comparison highlights the performance implications of executing a task using a serverless solution like OpenWhisk over present day alternatives.

The Docker solution basically attempts to mimic the underlying operation of OpenWhisk, with the idea of a single function running in a single container. Analogous to the task execution described for OpenWhisk, 16 functions are executed concurrently in distinct containers, with each one removed after completion. Bespoke container images needed to be built to include each function.

Comparable to the Docker solution and OpenWhisk, the native solution also executes 16 functions concurrently but without container-based virtualisation, meaning this solution does not suffer from the same overheads. Effectively, each function runs natively using libraries built-in to the virtual machine's operating system and concurrency is accomplished through manually executing each function on a separate thread. Theoretically, this solution should outperform the others.

As with all serverless solutions, tasks are performed through the execution of program functions. Hence, each of the experiment tasks require a counterpart in the form of a single coded function, written in Python which runs on all of the defined experiment execution methods.

Hypotheses. As a result of how these experiments are designed, a total of three hypotheses are formulated as predictions for each experiment's outcome. As such, each experiment tests one or more hypotheses and the results of each experiment proves or disproves them.

- **(H1**) In comparison to the other experiment tasks, the results from CPU experimentation on OpenWhisk, Docker and native solutions are most disparate.
- **(H2)** Experiments performed with the native solution yield lower runtimes and lower resource utilisation than the OpenWhisk and Docker solutions across all experiment tasks.
- **(H3)** Results from experimentation on OpenWhisk and Docker solutions are almost identical due to their shared architecture and functionality, though the Docker solution has a slight edge.

V. Performance Results

To guarantee accurate and reliable results, experiments were run five times, with five different iteration values appropriate to each experiment task. The perfomance results that are shown next represent the average of these runs.

A. CPU

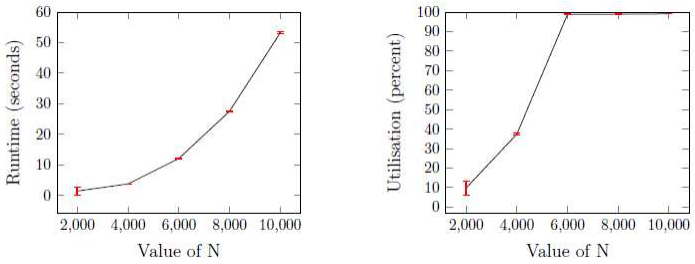

Figure 2. OpenWhisk CPU Runtime and Utilisation

To avoid issues with numbers becoming too large to sum or divide, values are converted to Decimal type with a precision of 50 decimal places. This is acceptable as the purpose of the experiment is not to achieve a particular value of Pi, but simply to stress the CPU.

As expected, Figure 2 depicts a gradual increase in runtime as the value of N, the number of iterations, increases. Apart from when N = 2000, standard deviations in runtime remained small which suggests that OpenWhisk is consistent in utilising the CPU. The standard deviations in CPU utilisation are also small, again with the exception ofwhen N = 2000. These anomalies in standard deviation

are clearly caused by the first run when N = 2000, and are a consequence of the initial creation of required containers. The trend appears to be linearithmic, which is the time complexity for the BBP formula. This indicates that OpenWhisk has little impact on the runtime of functions, not considering the time taken for initial startup.

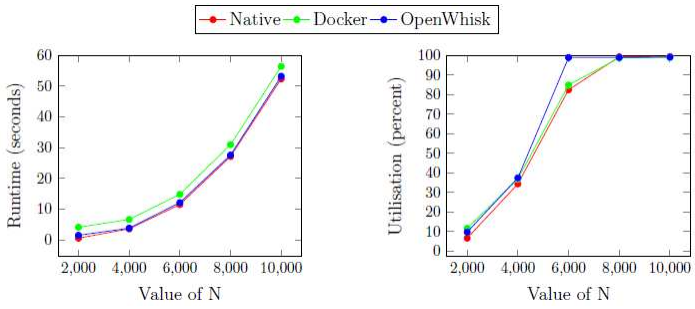

Figure 3. Benchmarking CPU Runtime and Utilisation

Recounting on the hypotheses made previously, Figure 3 depicts close runtimes between OpenWhisk, Docker and native solutions, though the Docker solution is slowest in all areas, disproving hypothesis H3. This is likely due to OpenWhisk's orchestration of containers being more optimal than what can be achieved through the Docker Command Line Interface. On the contrary, the CPU utlisation demonstrates OpenWhisk's being above that of the Docker and native solutions, implying that there is some truth in hypothesis H3. This disparity also partially confirms hypothesis H1.

B. Memory

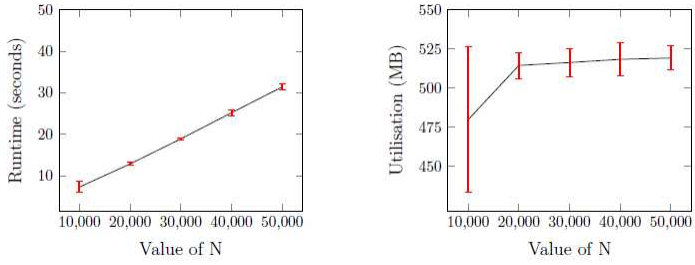

Figure 4. OpenWhisk Memory Runtime and Utilisation

The runtimes in Figure 4 appear to scale linearly with N (matrix size), and the standard deviation for each N is similar to those in the CPU experiment. However, while the results also show the largest standard deviation to be for the first value of N, it is caused by Run 2 and Run 5 instead. This is atypical to the expected behaviour of utilisation, which is for Run 1 to produce values above the average due to initial startup and container deployment. More unusual though, is the outcome that this large standard deviation arises from values below the average. One explanation for this is the utilisation measured here purely represents memory consumed by the functions being executed, not including amounts used by the underlying system that could be monopolising memory and possibly causing restrictions in allocation to OpenWhisk.

Figure 4 proves the linear increase in runtime as N increases as well as in utilisation. Despite this, the overlaps in standard deviations for utilisation show that memory allocation is variable.

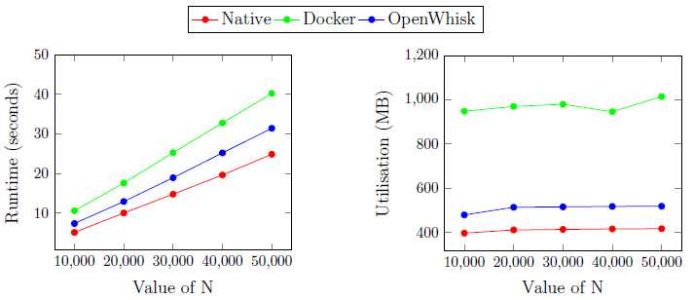

Figure 5. Benchmarking Memory Runtime and Utilisation

Disproving hypothesis H3, Figure 5 depicts the Docker solution as slower than OpenWhisk, also probably linked to container orchestration as mentioned for the CPU experiment. Similarly, H2 holds true for this experiment, with the native solution having the lowest runtimes and lowest utilisation. Despite the promising results from the CPU experiment, it seems that H1 is also disproven, as the results shown in Figure 5 for the memory experiment are more disparate. The Docker solution utilises significantly more memory in this experiment, which too could be linked to the inferiority this solution has when compared to OpenWhisk's orchestration capabilities.

C. Network

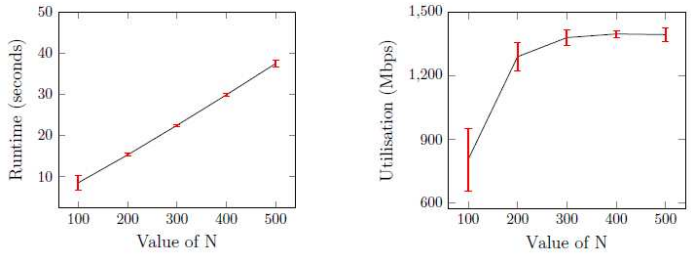

Figure 6. OpenWhisk Network Runtime and Utilisation

Reverting back to the same anomaly found in the CPU experiment runtimes, the standard deviation is large for the first value of N (number of HTTP requests) and caused by Run 1, as shown in Figure 6. Inline with the ongoing pattern, the largest deviation in network utilisation is found for the first value of N, though like the memory experiment, is caused by values much lower than the average.

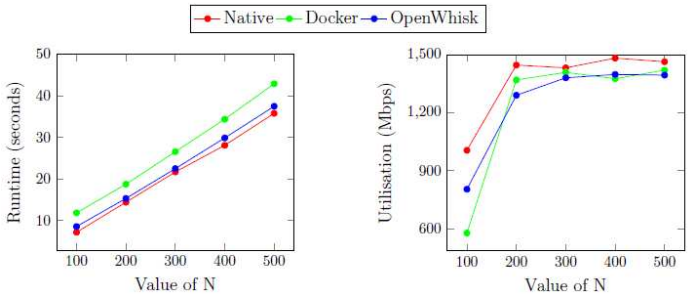

Figure 7. Benchmarking Network Runtime and Utilisation

For a third time, hypothesis H3 is disproved, as shown in Figure 7 where the Docker solution performed slower than OpenWhisk. Hypothesis H2 remains true for runtime, however utilisation is above the other solutions, which is actually a good thing. Based on that logic, H3 could be proven slightly as the Docker solution outerforms OpenWhisk for 3/5 of the different values of N, although the values are so close at each point that the two solutions' performances are practically the same. Also disproving H1, this experiment has a larger disparity in both runtime and utilisation when compared to the CPU experiments.

D. Discussion

As described in Section III, one of the motivations for evaluating Apache OpenWhisk is due to the lack of benchmarks and research, not only in the field of open-source serverless computing, but for OpenWhisk specifically too.

Considering the closest work on the assessment of OpenWhisk with respect to performance [26], [25], and given that neither of these papers evaluated OpenWhisk in similar ways, it is hard to draw conclusions around whether or not the findings in this investigation support or align with the findings in those. That said, the evaluation in [26] is closest in methodology as it experiments with CPU and memory intensive functions, using prime number computation and matrix multiplications respectively.

In essence, this investigation contributes further to research in this field. The experiments are designed to assess OpenWhisk's capabilities through performance intensive tasks and benchmark OpenWhisk in three performance areas against alternate methods, which mimic function execution performed by serverless computing.

VI. Conclusion and Future Work

This paper aimed to conduct a comprehensive evaluation of Apache OpenWhisk, principally focused on its performance. To do this, a series of experiments were designed and implemented to assess OpenWhisk's performance in different areas. The experiments involved creation and execution of program functions, each of which was intended to target and consume a specific hardware resource. Two metrics were of interest in experimentation: function runtime and resource utilisation. Together, these metrics would be used to demonstrate OpenWhisk's performance in terms of effectiveness and efficiency. Experiments also involved creation of two alternate solutions used as benchmarks for the results produced by OpenWhisk to provide some context and means for comparison. The results of each experiments showed that OpenWhisk could outperform a solution which employed similar functionality, through use of container-based virtualisation. It also demonstrated how close OpenWhisk is performance-wise to a more optimal solution which does not suffer from the overheads of virtualisation. In summary, this paper has contributed to existing research in the area of serverless computing by executing typical performance based experiments with unconventional real-world benchmarks.

There are many options for future work which could further add to this research topic:

- One of the extended requirements is to test OpenWhisk's performance in the area of concurrency. In particular, one could evaluate how OpenWhisk behaves in comparison to the same benchmarks.
- Other open-source serverless solutions could be deployed to provide a side-by-side comparison in performance on the cloud computing testbed.
- A qualitative evaluation of production serverless solutions with open-source solutions could be performed to measure trade-offs influenced by cost.

References

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the. ACM*, vol. 62, no. 12, p. 44–54, Nov. 2019.

[2] "Amazon web services. aws lambda," 2019, https://aws.amazon.com/lambda/.

[3] "Microsoft azure. azure functions," 2019, https://azure.microsoft.com/en-us/services/functions/.

[4] "Google. google cloud functions," 2019, https://cloud.google.com/functions.

[5] "Apache openwhisk. open source serverless cloud platform," 2019, https://openwhisk.apache.org/.

[6] "Fission - open source, kubernetes-native serverless framework," 2019, https://fission.io.

[7] "Ironfunctions - open source serverless computing," 2019, https://open.iron.io/.

[8] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," *CoRR*, vol. abs/1706.03178, 2017.

[9] E. van Eyk and A. Iosup, "Addressing performance challenges in serverless computing," in *Proceedings of ICT.OPEN 2018*. Amersfoort, The Netherlands: ACM, March 2018.

[10] "Nginx, inc. nginx," 2019, https://www.nginx.com/.

[11] "Apache. couchdb," 2019, http://couchdb.apache.org/.

[12] "Hashicorp. consul," 2019, https://www.consul.io/.

[13] "Apache. kafka," 2019, https://kafka.apache.org/.

[14] "Docker. empowering app development for developers," 2020, http://www.docker.com.

[15] V. Yussupov, U. Breitenbücher, F. Leymann, and M. Wurster, "A systematic mapping study on engineering function-as-a-service platforms and tools," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, Auckland, New Zealand, Dec 2019.

[16] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 442–450.

[17] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 133–145.

[18] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pp. 405–410.

[19] 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)*, Zurich, Switzerland, 2018, pp. 181–188.

[20] 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.

[21] M. Maas, K. Asanović, and J. Kubiatowicz, "Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 138–143.

[22] J. Spillner, "Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation," *CoRR*, vol. abs/1703.07562, 2017.

[23] S. K. Mohanty, G. Premsankar, and M. D. Francesco, "An evaluation of open source serverless computing frameworks," in *Proceedings of the 10th IEEE International Conference on Cloud Computing Technology and Science*. Nicosia, Cyprus: IEEE, 2018, pp. 115–120.

[24] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms," *Proceedings of the 5th International Workshop on Serverless Computing - WOSC '19*, 2019.

[25] T. Back and V. Andrikopoulos, "Using a microbenchmark to compare function as a service solutions," in *Proceedings of the 7th European Conference on Service-Oriented and Cloud Computing (ESOCC)*, K. Kritikos, P. Plebani, and F. de Paoli, Eds. Como, Italy: Springer, Sep. 2018, lecture Notes in Computer Science, Vol. 11116.

[26] A. Kuntsevich, P. Nasirifard, and H.-A. Jacobsen, "A distributed analysis and benchmarking framework for apache openwhisk serverless platform," in *Proceedings of the 19th ACM/IFIP International Middleware Conference*, Rennes, France, Dec. 2018.

[27] "Opennebula: Flexible enterprise cloud made simple," 2020, http://opennebula.org/.

[28] "Zabbix - an enterprise-class open source distributed monitoring solution," 2020, http://www.zabbix.com/.

[29] I. Molyneaux, *The Art of Application Performance Testing: From Strategy to Tools*. O'Reilly Media, 2015.

[30] D. Bailey, P. B. Borwein, and S. Plouffe, "On the rapid computation of various polylogarithmic constants," *Mathematics of Computation*, vol. 66, pp. 903—913, Apr 1997.