Performance Analysis of Apache OpenWhisk across the Edge-Cloud Continuum

Areej Alabbas*§, Ashish Kaushal†, Osama Almurshed*, Omer Rana*, Nitin Auluck†, Charith Perera*
*Cardiff University, United Kingdom
{alabbasam, almurshedo, ranaof, pererac}@cardiff.ac.uk
†Indian Institute of Technology Ropar, India
{ashish.19csz0003, nitin}@iitrpr.ac.in
§Imam Abdulrahman Bin Faisal University, Saudi Arabia

2023 IEEE 16th International Conference on Cloud Computing (CLOUD) | 979-8-3503-0481-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/CLOUD60044.2023.00054

***Abstract*—Serverless computing offers opportunities for auto-scaling, a pay-for-use cost model, quicker deployment and faster updates to support computing services. Apache OpenWhisk is one such open-source, distributed serverless platform that can be used to execute user functions in a stateless manner. We conduct a performance analysis of OpenWhisk on an edge-cloud continuum, using a function chain of video analysis applications. We consider a combination of Raspberry Pi and cloud nodes to deploy OpenWhisk, modifying a number of parameters, such as maximum memory limit and runtime, to investigate application behaviours. The five main factors considered are: cold and warm activation, memory and input size, CPU architecture, runtime packages used, and concurrent invocations. The results have been evaluated using initialization, and execution time, minimum memory requirement, inference time and accuracy.**

***Index Terms*—edge-cloud computing, serverless, function as a service, OpenWhisk, performance evaluation.**

I. Introduction

Serverless computing utilises containers and virtualization to deploy applications, to offer users with isolated environments that can be used to execute functions as code [1]. Users only pay for the computational resources (CPU time, memory utilisation etc) that are used for function execution. As a consequence, the use of serverless platforms leads to an increase in resource elasticity, seamless scalability and reduced operational expenses. Serverless computing platforms offered by cloud providers include AWS Lambda [2], Microsoft Azure Functions [3] and Google Cloud Functions [4]. However, these platforms can lead to vendor lock-in [5] and users are required to modify their functions due to limits on the size of function code, the time of execution, and the number of concurrent executions that can be performed using these commercial platforms [6]. Open source frameworks enable serverless computing to be executed on private infrastructure while avoiding vendor lock-in, such platforms include: Apache OpenWhisk [7], OpenFaaS [8], Fission [9] and Kubeless [10].

The number of contributors on the source-code repository identifies Apache OpenWhisk as the most widely used platform. Moreover, OpenWhisk also offers a lite version called Lean OpenWhisk [11], which can be deployed on resource-constrained edge devices. Therefore, we have chosen OpenWhisk in this study. Recent research on the performance of commercial and open-source serverless platforms mainly focuses on cloud-hosted platforms [12]–[19]. There are a very limited number of studies on edge serverless platforms [20]–[22]. Moreover, we also observe that the performance of OpenWhisk on edge and cloud resources using a real-life video analysis use-case has not been considered. We investigate the performance of OpenWhisk across different factors that contribute to the overall latency, including initialization time and execution time. We also investigate the impact of cold and warm activation, function input size, memory used, CPU architecture, runtime package and rate of concurrent invocations. The following are the key contributions of this paper: (i) deployment of OpenWhisk and Lean OpenWhisk over edge-cloud infrastructure. (ii) a distributed, Lean OpenWhisk framework that can be deployed on a cluster of RPi running ARM architecture. A custom Python based runtime for Lean OpenWhisk; (iii) performance analysis using a real-time video analytics application; (iv) analysis of latency, initialization time, memory requirement, inference time and accuracy of two models for object detection (TensorFlow and TensorFlow Lite) to improve model selection utilising edge-cloud serverless computing.

The rest of this paper is structured as follows: Section II reviews related works. Section III presents an overview of Apache OpenWhisk serverless platform. Section IV describes our video analysis application use case, and Section V evaluates OpenWhisk performance on edge-cloud infrastructure. Finally, Section VI summarizes our contributions.

II. Related Work

We divide related work into an evaluation of serverless platforms not including OpenWhisk and a more detailed discussion of studies that consider OpenWhisk specifically.

1) "Non-OpenWhisk serverless": Studies in this category have evaluated different serverless platforms, excluding OpenWhisk. Mohanty et al. [12] evaluate the auto-scaling and concurrent user capabilities of Fission, Kubeless, and OpenFaaS. Using Python applications, another study [13] analyzes latency across AWS, Google, and Microsoft serverless platforms. Lloyd et al. [14] identify five factors that affect AWS Lambda and Microsoft Azure latency.

2) "Cloud-based OpenWhisk": This category focuses on the performance of OpenWhisk on cloud platforms. Maissen et

2159-6190/23/$31.00 ©2023 IEEE
DOI 10.1109/CLOUD60044.2023.00054

al. [15] studied the influence of request rate, cloud location, memory size, and programming language on latency across different serverless providers. Djemame et al. [16] assessed OpenWhisk's effectiveness and efficiency on the cloud, comparing it to Docker and native function execution solutions. Back et al. [17] and Lee et al. [18] performed comparative analyses of OpenWhisk, AWS Lambda, Google Cloud, and Microsoft Azure, considering aspects like execution time, cost, and resource utilization. Kuntsevich et al. [19] evaluated OpenWhisk's limitations and bottlenecks on a private cloud.

3) "Hybrid edge-cloud and edge-based OpenWhisk": This category investigates OpenWhisk's performance on edge devices and hybrid edge-cloud setups. Javed et al. [20] compared OpenWhisk, OpenFaaS, and AWS Greengrass on edge resources, and AWS Lambda and Azure Functions on the cloud, measuring response time and success rate. Palade et al. [21] evaluated four open-source serverless platforms on edge including OpenWhisk, but not on resource-constrained devices like RPi. Tzenetopoulos et al. [22] evaluate OpenWhisk and OpenFaaS on a hybrid edge-cloud cluster with an emphasis on applications for optical character recognition. They deploy OpenWhisk on a single edge node and evaluate it in five steps.

Our study examines the performance of the OpenWhisk platform in edge-cloud environments. To do this, we explore the unexplored application of evaluating the function chain of video processing in an edge-cloud environment. We deploy OpenWhisk on multiple edge devices with multiple invokers, resulting in a complex and realistic configuration.

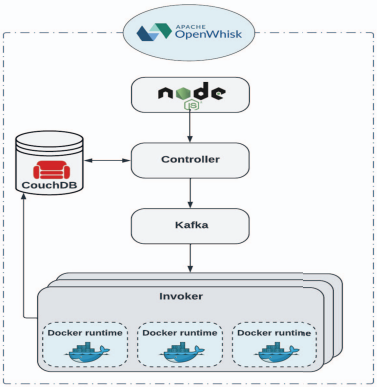


Fig. 1: Apache OpenWhisk architecture

III. Overview of Apache OpenWhisk

OpenWhisk is an open-source serverless cloud platform that allows users to execute functions in response to a trigger event. OpenWhisk utilises Function as a Service model (FaaS), where functions are referred to as actions and their invocations as activations. OpenWhisk assumes a direct relationship between memory and CPU limits of containers, so developers are only allowed to specify RAM (memory) size for executing their actions. Developers can create a chain of actions where one action calls another sequentially.The OpenWhisk architecture is shown in Figure 1 and has two key components: Controller and Invoker, both use Nginx [23], Kafka [24], Docker [25] and CouchDB [26]. Kafka connects and buffers messages between Controllers and Invokers. Lean OpenWhisk is designed for edge devices and is less resource-intensive than the full version. Kafka is replaced by an in-memory queue and the Invoker is co-located with the Controller.

IV. Use Case – Video Analysis Application

For our experimentation, we have considered a video analysis application as a realistic use-case scenario to investigate and analyze various crucial factors that affect the performance of OpenWhisk serverless platforms deployed on both edge and cloud resources. Most of the video analysis applications require processing of video streams in real-time. Moreover, these applications are sensitive to delay and require low latency response. To evaluate the application, we have fragmented our video analysis process into four interconnected and dependent functions. These functions are executed sequentially, with the output of each function serving as an input for the subsequent function in our chain. Partitioning the video analysis application allows us to efficiently deploy each function on a resource-constrained device in the edge layer. Moreover, because of the resource limitations imposed by platform providers (default memory limit for OpenWhisk is 256MB), these platforms are more suitable for executing smaller functions and tasks. A depiction of functions considered in the video analysis application is given in Figure 2.

Our video analysis application is composed of three stages: (i) **Pre-process**, (ii) **Analysis**, and (iii) **Result**. In the Pre-process stage, the video streams are loaded from the data source (camera) and decoded to extract individual video frames using the first function ***F1***. These extracted frames are then resized using ***F2*** function and sent to the next stage. In the Analysis stage, a real-time TensorFlow [27] based object-detection algorithm is applied on each frame to detect the desired objects using function ***F3***. The last stage, Result, extracts the detected objects by drawing boxes around them and adding labels on each object. The output frames are then uploaded to the database (final destination) using function ***F4***. Functions *F1* and *F2* are considered light functions while functions *F3* and *F4* have computational resource requirements. The computational level of our benchmarking functions, outlined in Table I, is tied to the resources used for function execution. Low level involves minimal arithmetic operations, such as resizing an image matrix. High level requires extensive operations and memory usage for storing intermediate results, such as large matrix manipulations. Medium level requires moderate operations like image slicing, cropping, and data storage via an input-output streaming channel.

Function ID	Function Name	Computational level
F1	Decod&Extract Frame	Low
F2	Resize Frame	Low
F3	Object Detect	High
F4	Object Extract & Save	Medium

TABLE I: List of functions and their computational levels

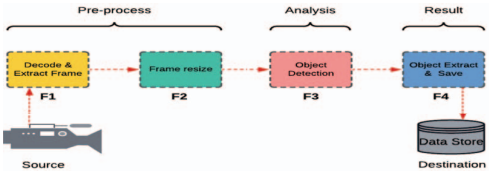


Fig. 2: Video Analysis Application

In stage (ii) of our video analysis application, we have implemented functions F3 and F3_PRIME. F3 utilizes TensorFlow and F3_PRIME uses TensorFlow Lite. MobileNetV2 SSD [28] and MobileNetV1 SSD [29] models, both trained on the Common Objects in Context (COCO) dataset [30], were used for inference in F3 and F3_PRIME, respectively. TensorFlow Lite is a compact and lightweight variant of TensorFlow, with lower accuracy but higher execution performance.

V. Implementation and Evaluation

The integration of our edge-cloud distributed framework with the designed testbed is shown in given Figure 3. The setup includes VMs deployed on OpenStack cloud platform, resource-constrained RPi as edge devices, and video capture via cameras as a data source. The detailed hardware and software specifications of edge and cloud nodes are shown in Table II and Table III respectively.

Location	No. of Nodes	Processor	Architecture	Cores	RAM
Cloud (VMs)	3	Intel Xeon	x86_64 GNU	2(vCPUs)	4GB
Edge (RPis)	6	Cortex-A72	armv7l GNU	4(vCPUs)	4GB

TABLE II: Hardware specifications of cloud and edge nodes

Software	Edge	Cloud
OS	Raspbian GNU/Linux 11 (bullseye)	Ubuntu 20.04.3 LTS (Focal Fossa)
OpenWhisk	incubator-openwhisk(Lean version)	1.0.0 (full version)
WSK CLI	0.10.0-incubating	v1.2.0
Ansible	2.7.9	-
Helm	-	v3.9.0
Kubernetes	-	1.20.15
Docker	20.10.16	20.10.12
Python	3.7 &3.9	3.7 &3.9
OpenCV	4.6.0-dev (Lite version)	4.6.0
TensorFlow	Lite & full(2.2.0)	Lite & full(2.9.1)

TABLE III: Software specifications for cloud and edge nodes

On cloud, we have deployed OpenWhisk as a Kubernetes cluster on three VMs. All VMs run Ubuntu Server 20.04 OS, and have 2 cores of CPUs and 4GB RAM; two of them are used to host Invokers, while the third one is utilised to host the remaining OpenWhisk components.

On edge, we have used six Raspberry Pi's 4 computer - model B, running Raspberry Pi OS Lite (32-bit) Debian Bullseye. Each RPi has a 1.5GHz 64-bit quad-core CPU (ARM processor) and 4GB RAM. We have deployed Lean Apache Open Whisk on edge devices that do not require Kafka and Invokers as separate entities. However, Lean OpenWhisk is not natively compatible with ARM architecture; only the x64 and x86 architectures are supported by the OpenWhisk platform, so in order for it to be compatible, we have customized Docker images of the platform for setting up Lean OpenWhisk on RPi

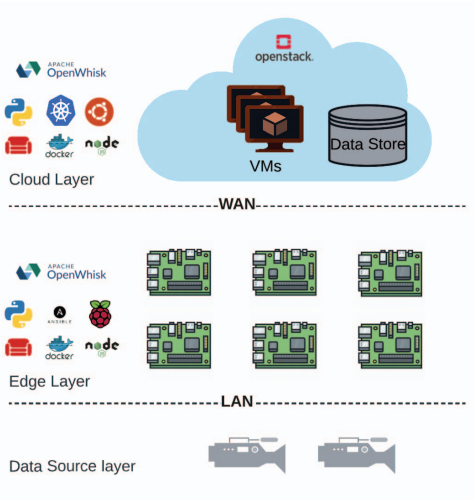


Fig. 3: Overview of cloud-edge infrastructure

devices with the ARM architecture. A Python runtime for Lean OpenWhisk running on ARM architecture devices has also been implemented; currently Lean OpenWhisk has runtime support in only one language i.e.NodeJS-6, for ARM devices. Different docker images have been created as runtime for our function invocations using Python programming language with OpenCV, TensorFlow Full, and TensorFlow Lite versions. We use 4 RPi devices as Invokers and the rest are utilised for deploying the Nginx and CouchDB modules. To ensure that no other components are using the system resource, each Invoker is deployed separately on a different machine.

Moreover,We have installed the WSK CLI tool. This allows users to easily create, update, and invoke functions within our system. In addition, as users of OpenWhisk can specify only a single dimension (memory requirement), we have made modifications to the default settings of OpenWhisk by increasing the maximum memory limit for function execution on all Invoker machines to 3072MB. We have increased the timeout limit for function execution to 300000ms (5 mins). The motivation behind increasing timeout limit is driven by high computational requirements of designed function F3.

A. Proposed Methodology

Our proposed approach is designed to analyze the performance of edge-cloud framework in terms of latency. We executed the designed functions in both synchronous and asynchronous invocations mode on both edge, cloud resources and compared their performance with each other. The implications of various critical factors that directly influence function execution is also considered in this work. The five main factors investigated here are:

Cold and Warm Activation: In serverless platform, the delays incurred during the cold activation include the time to initialize the docker container runtime and time to execute function's code. However, in warm activation, there is no initialization time. The duration of a cold activation can

vary depending on several factors, such as the programming language used, size of the function's code and package dependencies, and the resources required to run the function. Thus, this step compares the latency of cold and warm activation for functions with different package dependencies (e.g OpenCV, TensorFlow). In addition, we created a metric called *Activation Ratio* which calculates the ratio between the Warm and Cold activation. This ratio is utilised to decide whether to keep the warm containers operational or not.

Resolution Type	Common Name	Frame Size(Pixel)
LD (Low Definition)	240p	320 x 240
SD (Standard Definition)	480p	640 x 480
HD (High Definition)	720p	1280 x 720
FHD (Full High Definition)	1080p	1920 x 1080

TABLE IV: List of Video Frame Resolutions

Memory Setting and Input Size: The execution time of a function on serverless platforms is influenced by two crucial factors, namely, the amount of memory allocated and the size of the input provided. Therefore, we run our video analysis functions using different frame resolutions (frame sizes) as shown in Table IV and using different memory settings ranging from 128MB to 3072MB. Furthermore, we measure the minimum memory capacity necessary to run each function without failure.

CPU Architecture: The performance of distinct functions tends to vary based on the resources utilised, such as RPi or cloud. However, resource-constrained devices like RPi devices can result in a higher latency or delay when executing these functions. As the next step of our proposed method, we measure the total latency and docker initialization time of our functions using different resources RPis and cloud servers.

Runtime Package: In OpenWhisk, different runtimes (e.g Python, Java etc) and runtime's packages (e.g OpenCV and TensorFlow) can impact the total latency of a running function. As mentioned earlier in section IV, we implemented two versions of the third function, F3 and F3 PRIME. The difference between them is the runtime package used for the object detection process. The two packages used for evaluation are: TensorFlow and TensorFlow Lite. We have measured the inference time, and compared accuracy of these two modules. To measure the accuracy in our experiment, we considered four metrics: (1) Object Location: whether the object in image is properly (x and y axis aligned) detected or not; (2) Class Labels: whether the detected object is labeled correctly or not; (3) Detection Confidence: the percentage of confidence with which the object has been detected; and (4) Detection Count: number of objects that have been totally detected in the image.

Concurrent Invocations: A serverless function in a video analysis application may be invoked concurrently several times, depending on the frame rate (e.g., 15 or 25 frames per second). Consequently, any bottlenecks in the various components of the serverless platform may increase the overall latency of function execution. To address this issue, we conducted performance evaluations of our serverless platform using different concurrent invocation rates(e.g., 1, 5, 10, and 15) to assess the impact of concurrency on its overall performance. Contrary to all previous factors that utilized synchronous invocations, in this factor, we conducted invocations in an asynchronous manner.

The Total Latency (TL) for running the function in OpenWhisk consists of (i) Initialization Time (T_{init}), the time to initialize docker container runtime (for warm activation:- $T_{init} = 0$) and (ii) Execution Time (T_{exec}), the time to execute function. Therefore, the Total Latency can be mathematically represented as: $TL = T_{init} + T_{exec}$. We have benchmarked the Initialization Time, Execution Time, Total Latency, activation ratio, minimum amount of resource required (memory), and inference time and accuracy of TensorFlow and Tensorflow Lite on both edge and cloud resources.

B. Results

*1) **Impact of Cold and Warm Activations**:* Figure 4 shows the total latency in cold and warm activations on RPi. In cold activation, the average latency increased by 3x, 4x and 2x for F1, F2, and F4 respectively compared to warm start. Figure 5 illustrates the initialization time for executing each function on RPi. F1 and F2 have similar average initialization time (approx. 1500ms), using the same Docker image consisting of Python, OpenCV Lite packages for execution. F3 and F4 however require installing TensorFlow packages in the Docker runtime, therefore the average initialization time is much higher (around 10000ms) compared to F1, F2.

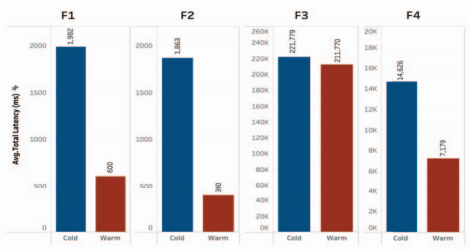


Fig. 4: Total latency in Cold and Warm activation on RPi.

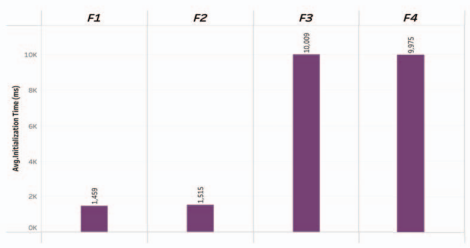


Fig. 5: Initialization time on RPi

From Figures 4 and 5, docker initialization exceeds execution time ($T_{exec} = TL - T_{init}$) for F1, F2, F4 whereas for F3 the execution time exceeds initialization time. It is realised that functions with an activation ratio higher than 1 perform better with warm containers, whereas functions with an activation ratio below or equal to 1 perform similarly in both cold and warm activation.

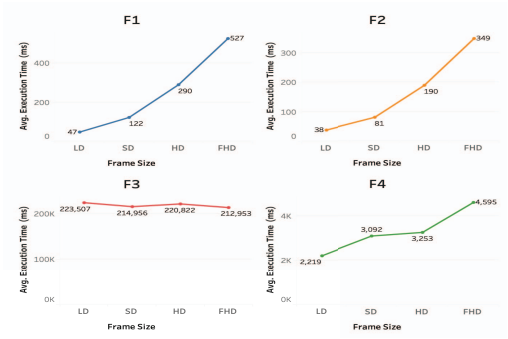


Fig. 6: Execution Time for the different frame sizes on RPi

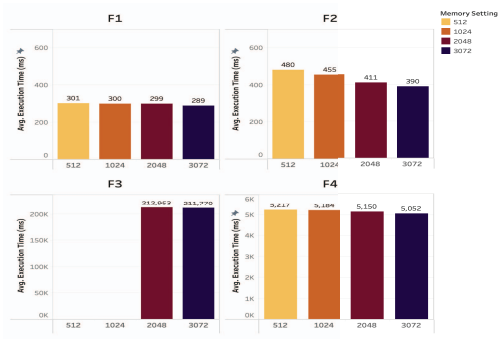


Fig. 7: Execution Time for different memory settings on RPi

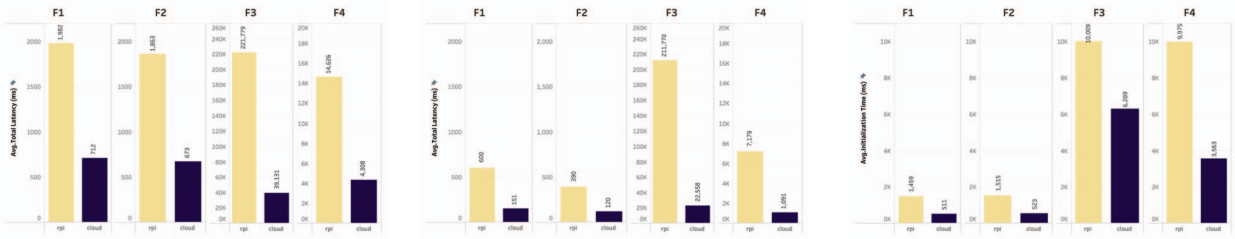
*2) **Impact of Input Size and Memory Setting**:* Figure 6 and 7 show the execution time of four functions running on RPi nodes with different frame sizes and different memory settings respectively. The overall average execution time for F1, F2, and F4 increases when the input size for each function is increased. For function F3, the execution time did not increase with the increase in frame size because a resized frame from the previous function (F2) was utilised as input in this function.

Secondly, an increase in memory setting does not significantly impact the average execution time of functions as seen in Figure 7. This is due to the fact that Docker's interaction with OpenWhisk permits partially dynamic memory allocation that restricts utilisation of total memory space during function execution. Similar results has been mentioned in works [17], [31]. In order to determine the minimum amount of memory required for running each function without failure, we have tested each function using six different memory settings starting from 128MB to 3072MB using the same input size. The result shows that F1 and F2 are lighter functions and can be run with a small amount of memory (from 128MB) while F3 and F4 are heavier functions and need more intensive resources with a minimum of 2048MB and 512MB respectively.

*3) **Impact of CPU Architecture**:* Figures 8a and 8b illustrate the latency of the four functions executed on RPi and cloud, using both cold and warm activations respectively. By running functions on cloud node, the total latency of F1 and F2 decreased by 2.7x and 2.6x for cold activation and by 3.9x and 3.2x for warm activation. For F3 and F4, the latency decreased by up to 5.6x and 3.3x in the cold invocations whereas it decreased up to 9.3x and 6.5x in the warm invocations, when executed on cloud. As shown in Figure 8c, the time to initialise a docker container on RPi devices is higher than on the cloud. The initialization time of the first two functions increased by 2.8x, and for F3 and F4 it increased by 1.5x and 2.7x on RPi.

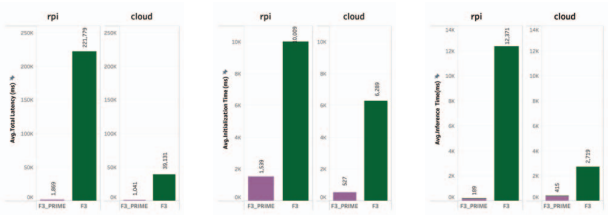
*4) **Impact of Runtime Packages**:* Figures 9a and 9b show the total latency and the initialization time of F3 and F3_PRIME on both RPi and Cloud. It is observed that total latency for F3_PRIME (in comparison to F3) is reduced by almost 118x and 37x times on RPi and Cloud respectively. F3_PRIME has a lower initialization time than F3 on both Cloud and RPi. The performance of full and lite versions are bound to the CPU resource architecture and the type, size of data which affect data locality during arithmetic operations, i.e., data in registers, cache, or main memory. Although the RPi has more CPU cores, it requires additional CPU instructions that shift data across memory tiers, e.g. cache to registers, to overcome the limitation of the fast memory – float-32 (8bytes) is used in the full version, whereas integer data type (int-8, 1 byte), is used in the lite version. Converting float-32 to int-8 is known as *quantization*, as used in the *lite* version.

One of the significant results observed is that: running F3_PRIME on RPi has a lower delay (with over 100ms) than on the Cloud in warm activation (Figure 10). This is due to the fact that lite version requires less memory space near the CPU (solving the RAM limitation issue for 32-bit) and that the RPi has four cores that run more operations in parallel.



(a) Cold activation latency (b) Warm activation latency (c) Initialization time

Fig. 8: Comparison: RPi and Cloud



(a) Total latency (b) Init. time (c) Inference time

Fig. 9: Comparison: F3 and F3_PRIME

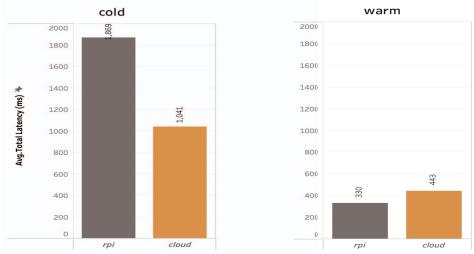
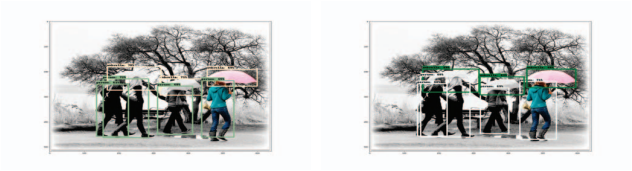


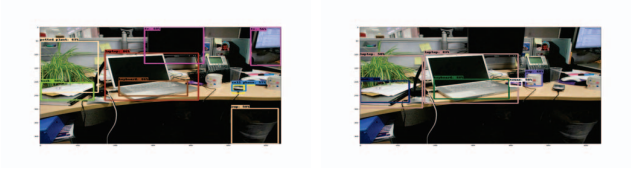
Fig. 10: Average latency: F3_PRIME in cold/warm activation

F3_PRIME can run with a minimum of 128MB, while F3 requires 2048+MB for function execution. Quantization reduces memory footprint of TensorFlow Lite. The inference time of both functions is illustrated on Figure 9c, showing the inference time for the lite version on Cloud is double compated to the RPi. The 32-bit system of RPis requires additional CPU instructions for floating-point computations and data transfers via the memory hierarchy. In the cloud, quantization reduces loading times and float operations, but has a minimal effect on data transfers. Further performance optimization on a 64-bit cloud produces less significant results compared to RPis. Architectural variations such as processor type and data locality can cause bottlenecks in shared-memory parallel computing, leading to performance disparities [32]. Although both models predicted the right object classes, in the right position, as seen in Figure 11, the confidence score for detecting an object is higher in Figure 11a compared to Figure 11b. This is because lite models use a quantization technique on model parameters, trading-off generalization of model with inference time and memory. Moreover, in Figure 11c and 11d, having a higher confidence score, we can see that the full version was able to detect a greater number of objects. In general, we can say that the full version performs better than lite version but requires more time and resources for execution.

*5) **Impact of Concurrent Invocations**:* Figure 12 shows that with increase in the number of concurrent invocations, the overall delay on both edge and cloud nodes also increases. We concurrently execute 1, 5, 10 and 15 functions on both cloud and edge layers asynchronously. Concurrent executions was performed with function F1 only, as it has minimum memory (128MB) requirement for execution. However, for function F3 the minimum memory requirements are 2048MB. This does not allow us to run multiple invocations due to maximum memory limitation of the Invoker machines (3072MB).

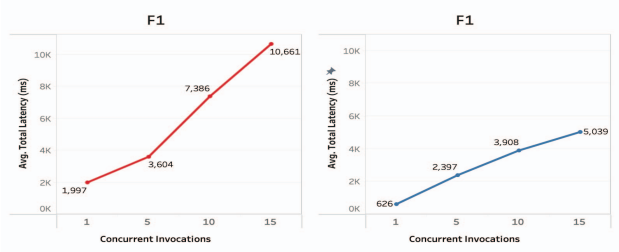


(a) Full version (b) Lite version



(c) Full version (d) Lite version

Fig. 11: Accuracy comparison of TensorFlow Full and TensorFlow Lite



(a) RPi (b) Cloud

Fig. 12: Average latency: concurrent invocations

VI. Conclusions

A video processing application is used to benchmark an open source serverless platform using TensorFlow and TensorFlow Lite. We consider both cold and warm activation, function input size and memory, CPU architecture, runtime package and rate of concurrent invocations. We observe that OpenWhisk has a high cold activation latency during function execution. Memory allocated and input size are two crucial factors that affect function execution time. Increase in memory does not significantly impact average function execution time, however increasing the input size has an impact. We observe that executing functions on cloud nodes (compared to RPi nodes) reduces total latency in both cold and warm activation. However, executing these functions on cloud could increase network delay, subsequently impacting total latency. The total latency, execution time, initialization time and inference time for the functions using TensorFlow Lite is low compared to the full TensorFlow package. Concurrent executions improve resource utilisation on both cloud and RPi nodes; however, this increases average latency significantly. Concurrent execution on cloud node is recommended as RPi takes almost 2x more time to execute a similar number of functions.

REFERENCES

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

[2] Aws lambda. [Online]. Available: https://aws.amazon.com/lambda/

[3] Microsoft azure functions. [Online]. Available: https://azure.microsoft.com/en-us/products/functions/

[4] Google cloud functions. [Online]. Available: https://cloud.google.com/functions

[5] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*. Springer, 2017, pp. 1–20.

[6] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 162–169.

[7] Apache openwhisk. open source serverless cloud platform. [Online]. Available: https://openwhisk.apache.org

[8] Openfaas. [Online]. Available: https://docs.openfaas.com

[9] Fission open source kubernetes-native serverless framework. [Online]. Available: https://fission.io

[10] Kubeless. [Online]. Available: https://github.com/vmware-archive/kubeless

[11] D. Breitgand. Lean openwhisk: Open source faas for edge computing.

[12] S. K. Mohanty, G. Premsankar, M. Di Francesco *et al.*, "An evaluation of open source serverless computing frameworks." *CloudCom*, vol. 2018, pp. 115–120, 2018.

[13] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 502–504.

[14] 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)*. IEEE, 2018, pp. 159–169.

[15] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "Faasdom: A benchmark suite for serverless computing," in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, 2020, pp. 73–84.

[16] K. Djemame, M. Parker, and D. Datsev, "Open-source serverless architectures: an evaluation of apache openwhisk in: 2020 ieee/acm 13th international conference on utility and cloud computing (ucc), 329–335," *DOI: https://doi. org/10.1109/UCC48980*, 2020.

[17] T. Back and V. Andrikopoulos, "Using a microbenchmark to compare function as a service solutions," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2018, pp. 146–160.

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

[19] A. Kuntsevich, P. Nasirifard, and H.-A. Jacobsen, "A distributed analysis and benchmarking framework for apache openwhisk serverless platform," in *Proceedings of the 19th International Middleware Conference (Posters)*, 2018, pp. 3–4.

[20] H. Javed, A. N. Toosi, and M. S. Aslanpour, "Serverless platforms on the edge: a performance analysis," in *New Frontiers in Cloud Computing and Internet of Things*. Springer, 2022, pp. 165–184.

[21] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642. IEEE, 2019, pp. 206–211.

[22] A. Tzenetopoulos, E. Apostolakis, A. Tzomaka, C. Papakostopoulos, K. Stavrakakis, M. Katsaragakis, I. Oroutzoglou, D. Masouros, S. Xydis, and D. Soudris, "Faas and curious: Performance implications of serverless functions on edge computing platforms," in *International Conference on High Performance Computing*. Springer, 2021, pp. 428–438.

[23] Nginx. [Online]. Available: https://www.nginx.com

[24] Kafka. [Online]. Available: https://kafka.apache.org

[25] Docker. [Online]. Available: https://www.docker.com

[26] Couchdb. [Online]. Available: https://couchdb.apache.org

[27] Tensorflow. [Online]. Available: https://www.tensorflow.org

[28] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[29] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[30] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 2014, pp. 740–755.

[31] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking heterogeneous cloud functions," in *Euro-Par 2017: Parallel Processing Workshops: Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers 23*. Springer, 2018, pp. 415–426.

[32] O. Almurshed, P. Patros, V. Huang, M. Mayo, M. Ooi, R. Chard, K. Chard, O. Rana, H. Nagra, M. Baughman *et al.*, "Adaptive edge-cloud environments for rural ai," in *2022 IEEE International Conference on Services Computing (SCC)*. IEEE, 2022, pp. 74–83.