Performance Evaluation of Serverless Edge Computing for Machine Learning Applications

Quoc Lap Trieu  
Western Sydney University  
Sydney, Australia  
0000-0003-0678-374X

Bahman Javadi  
Western Sydney University  
Sydney, Australia  
0000-0003-2351-9801

Jim Basilakis  
Western Sydney University  
Sydney, Australia  
0000-0002-7440-1320

Adel N. Toosi  
Monash University  
Melbourne, Australia  
0000-0001-5655-5337

Abstract—Next generation technologies such as smart health-care, self-driving cars, and smart cities require new approaches to deal with the network traffic generated by the Internet of Things (IoT) devices, as well as efficient programming models to deploy machine learning techniques. Serverless edge computing is an emerging computing paradigm from the integration of two recent technologies, edge computing and serverless computing, that can possibly address these challenges. However, there is little work to explore the capability and performance of such a technology. In this paper, a comprehensive performance analysis of a serverless edge computing system using popular open-source frameworks, namely, Kubeless, OpenFaaS, Fission, and funcX is presented. The experiments considered different programming languages, workloads, and the number of concurrent users. The machine learning workloads have been used to evaluate the performance of the system under different working conditions to provide insights into the best practices. The evaluation results revealed some of the current challenges in serverless edge computing and open research opportunities in this emerging technology for machine learning applications.

Index Terms—Edge computing, Serverless computing, Machine learning, Response time, Autoscaling.

I. INTRODUCTION

Machine Learning (ML) applications incorporate data-driven, actionable insights into the user experience and gradually are becoming part of our daily life [1]. From intelligent traffic control and autonomous vehicles to video surveillance and smart healthcare, machine learning applications enable users to more efficiently complete a desired task or action. What all these applications have in common is the need for advanced data analytics and machine learning models, which are normally hosted in cloud computing infrastructures. The drawback of the cloud is high communication latency between the cloud and the points where data is generated by the Internet of Things (IoT) devices, or where the analysis is needed [2]. To address this challenge and provide a better quality of experience for users, edge computing has emerged, in which computing and storage nodes are placed at the network’s edge in close proximity to users [3].

Having data analytics and machine learning on the edge rather than in the cloud forces developers to resort to ad hoc solutions specifically tailored to the available infrastructure. The process is heavily manual, task-specific and error-prone, and usually requires good knowledge of the underlying infrastructure [4]. Consequently, when faced with large-scale, heterogeneous resource pools, performing effective machine learning is difficult. Thus, there is a need for new programming models and platforms to develop machine learning applications with less operational complexity for the edge. In order to address this challenge in the new edge-cloud computing era, serverless computing [5] appears to be a promising solution.

Serverless computing emerged as a solution for the complexity of cloud platforms and consists of services where users only provide the computing code that needs to be executed, and cloud providers manage all the infrastructure and platforms that enable code execution [6]. So, having serverless model for developing machine learning applications for edge-cloud computing environments could improve the productivity of the application developers. While serverless architecture is becoming popular in cloud computing [7], this model is yet to be explored and evaluated for edge computing. In fact, there is a limited number of works on serverless edge computing which mainly focused on conceptual models, architectural challenges and open issues [8]–[10].

In this paper, a comprehensive performance analysis of a serverless edge computing system using popular open-source frameworks, namely, Kubeless,¹ OpenFaaS,² Fission,³ and funcX⁴ is presented. The key contributions of this paper are as follows:

- Extensive performance evaluation of serverless edge computing’s latency considering different programming languages, workloads, and the number of concurrent users;
- Investigation of autoscaling behaviour for serverless edge computing under different machine learning workloads to provide insights into the design choices;
- Analysis of training time and inference time for machine learning workloads for better model selection for serverless edge computing.

II. SYSTEM OVERVIEW

In this section, we first review our selected open-source serverless frameworks, and then we present the system architecture for evaluation of these serverless frameworks in the edge computing configuration.

¹Kubeless, https://github.com/vmware-archive/kubeless.
²OpenFaaS, https://github.com/openfaas/faas.
³Fission, https://github.com/fission/fission
⁴funcX, https://github.com/funcx-faas/funcX
### A. Open serverless frameworks

In this subsection, we describe four open source serverless frameworks, namely OpenFaaS, Kubeless, Fission, and funcX. We chose these frameworks due to their popularity and distinct features as listed in Table I. OpenFaaS, Kubeless, and Fission utilize container orchestration to manage the networking and lifecycle of the containers, whereas funcX may be deployed with or without container orchestration. Please refer to [11] to read about the details of these frameworks.

### B. System implementation

In order to evaluate the selected frameworks, we set up a serverless edge computing system using a cluster of Raspberry Pis as depicted in Fig 1. There are four Raspberry Pis 4 Model B with a 1.5GHz 64-bit quad-core ARM processor and 4GB memory running Ubuntu 20.04.3 LTS. We also utilize Microk8s\(^5\) as the container orchestration, which is a lightweight distribution of Kubernetes built for IoT and edge computing devices. The Kubernetes cluster can also access to a cloud storage (i.e., AWS S3) to download the required files and models for the system workload.

As illustrated in Fig 1, all four serverless frameworks are deployed separately on the Raspberry Pis cluster which is managed by Kubernetes. The cluster nodes are interconnected using a local WiFi network. For the sake of accuracy, the experiments are run on an isolated network to reduce the impact of network overheads. The system workload and requests are generated by a test machine running Apache JMeter\(^6\) v5.4 on the same local network to trigger HTTP requests that invoke functions deployed on each serverless framework. This process is performed through a distributed load testing procedure and orchestrated using the test machine that has a JMeter client installed. The test machine is a laptop running macOS with 2.3GHz 8-core Intel Core i9 and 16GB RAM. The master node of the Kubernetes cluster is also located in the test machine.

---

### III. PERFORMANCE EVALUATION

In this section, we provide the performance evaluation results and discussions on various serverless frameworks under different workloads. The main performance metric is response time. The response time consists of the time it takes a request to reach the edge device, the time to execute the function on the device, and the time it takes the response to arrive to the JMeter client. Other metrics such as success rate and number of pods are also reported for some experiments.

#### A. Experimental setup

We configure JMeter to generate HTTP requests that invoke the functions deployed on each framework. The JMeter tool is set up to send 1000 requests with different levels of concurrency (\(n = 1, 5, 10, 15\) and 20). These concurrent requests/users affect the number of simultaneous requests received by the framework and are determined based on the computational power available in the cluster. We also measure the impact of auto-scaling on response time for different workloads. The CPU utilization is used as a metric to perform the auto-scaling, and is set to 50% for all frameworks in all experiments. When utilization exceeds this threshold, the

---

\(^5\)Microk8s, https://microk8s.io/.

\(^6\)JMeter, https://jmeter.apache.org/.

---

**TABLE I: Characterisations of the considered serverless frameworks**

| Features                  | OpenFaaS                          | Kubeless                          | Fission                          | funcX   |
|---------------------------|-----------------------------------|-----------------------------------|----------------------------------|---------|
| Supported Languages       | Python, C#, Go, Node.js, Ruby and custom containers | Python, Node.js, Ruby, PHP, Go, Java, .NET and custom containers | Python, Go, Node.js, Ruby, Perl, Bash, .NET, PHP and custom containers | Python  |
| Container orchestration   | Kubernetes, Docker, Swarm         | Kubernetes                        | Kubernetes                       | Singularity, Shifter, Docker |
| Auto scaling metric       | CPU utilization, QPS and custom metrics | CPU utilization, QPS and custom metrics | CPU utilization, QPS            | QPS     |
| Triggers                  | HTTP, event, schedule             | HTTP, event, schedule             | HTTP, event, schedule           | HTTP, GlobusAutomate |
| Maximum Walltime          | NA                                | NA                                | Undefined                       | No limit |
| Message queue             | NAIS, Kalka                       | NAIS, Kalka                       | NAIS, Azure storage queue       | Redis    |
| Monitoring tool           | Prometheus                        | Prometheus                        | Istio                           | funcX agent |
| State                     | Stateless                         | Stateless                         | Stateless                        | State-full |
| Development Language      | Go                                | Go                                | Go                              | Python   |
| Licences                  | MIT                               | Apache 2.0                        | Apache 2.0                      | Apache 2.0 |

---

**Fig. 1: Serverless edge computing implementation.**
TABLE II: Workload configuration

| Workload | Content                      | Language | Configuration          |
|----------|------------------------------|----------|------------------------|
| Helloworld | Return request               | Python and Node.js | Steps: 128; 1D CNN + Pooling: 4; Dense: 1 |
| CNN      | Inference for a CNN model    | Python   | Steps: 128; 1D CNN + Pooling: 4; Dense: 1 |
| LSTM     | Inference for a LSTM model   | Python   | Steps: 128; Size: 27; LSTM layers: 2, Dense: 1 |

creation of a new pod is triggered. All frameworks, except funcX adopt the Kubernetes’ Horizontal Pod Autoscaler to perform scaling based on the CPU utilization.

We used three different application workloads, including a simple Helloworld function as well as two machine learning models based on Convolutional Neural Networks (CNN) and Long short-term memory (LSTM). These workloads and their configurations are listed in Table II. The Helloworld function written in Node.js and Python and receives HTTP request and replies with a confirmation message. This function is used to measure the overhead of each framework for two common programming languages. Each experiment is repeated 3 times in order to maintain statistical accuracy.

CNNs and LSTMs are common deep learning models that have widely been used in image, speech and text recognition, and characteristically use a set of common computations. The CNN has additional sparsely connected convolutional layers compared to its subsequent fully connected dense layers that are optimised for capturing spatial and temporal inputs. In this paper, a one dimensional CNN (Conv1D) is used that specialises in sequences and time-series data. The LSTM consists of a series of gates that unroll programmatically in a sequence that attempts to also model time or sequence-dependent behaviour. The gates within the LSTM cells control information flow within the data sequence, determining whether information is to be remembered or forgotten.

The machine learning workload consists of classification on human activity recognition tasks using smartphone accelerometer and gyroscope data available from the UCI repository. There are over 10k instances of data consisting of nine channels of input and six activity tasks to be recognised in the output. The CNN and LSTM models used for training and classification of the time-series data are adopted from the open source Human Activity Recognition (HAR) project in Python. The CNN model is trained using 1000 epochs in batches of 600, while LSTM model training requires only 15 epochs that is unbatched. Training has been conducted on the test machine (x86) and repeated 3 times resulting in the training times and accuracy shown in Table III. On completion of training, the model weights are uploaded to the cloud storage as illustrated in Fig. 1, which are subsequently downloaded only once by edge devices to execute the workloads on either CNN or LSTM activity classification. A remote procedure call (RPC) from a client program running on JMeter is used to simulate sending of an input stream of human activity data to the machine learning algorithms deployed on the framework,

that will respond with an output of the classified activity.

B. Impact of concurrent users

In this experiment, we measure the response time of Helloworld function for different levels of concurrency in both Python and Node.js for all serverless frameworks. Since funcX does not support Node.js, we only report results of the Python function for this framework. We used Helloworld function to have minimal overhead in terms of the function logic and its dependencies. We deploy the function on each framework and invoke it through HTTP. We disable autoscaling and run a single pod per edge device with a fixed number of concurrency in each experiment. By doing so, we avoid possible impact of autoscaling on response times when scaling in/out functions, i.e., when creating new function pods.

The result of this experiments is plotted in Fig. 2 which shows the average response time of each serverless framework for both Python and Node.js functions. The lowest average response time is achieved by Kubeless in all scenarios. We observe that Kubeless and OpenFaaS maintain a response time below 50 ms and 800 ms across all scenarios for Node.js and Python, respectively. We also note that the response times have steady increases with the number of concurrent users for Kubeless and OpenFaaS. However, for Fission and funcX, there is a higher increase in response time with the number of users which is related to the architecture of these frameworks to dispatch users’ requests to available pods. We also observed that average response time for Node.js is considerably lower than Python function except with Fission which are about the same. The speed up ratio (Python/Node.js) is increased by the number of concurrent users and it reaches 10.3 for Kubeless for 20 concurrent users and 19.9 times for OpenFaaS for 15 concurrent users.

Fig. 3 shows the cumulative distribution function of the response time (on log scale) for 5 concurrent users for all frameworks in both Python and Node.js. As can be seen, the system performance while running Node.js function is quite stable for both Kubeless and OpenFaaS which reveals the suitability of this language for latency-sensitive applications. For Python function, the system performance is less stable for the higher number of concurrent users with long-tail distributions for the response time as depicted in these figures. In addition, the performance gap between Kubeless and other frameworks is much bigger for the Python function in compare with the Node.js function. For both funcX and Fission, the response time is slightly higher than other frameworks which is increased by the number of concurrent users. We also noticed that for one user, the response time of Fission

7UCI, https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones.
8HAR, https://github.com/bhimmetoglu/time-series-medicine.
9Workload, https://github.com/SDC-Lab/ServerlessWorkload
is comparable to OpenFaaS but this is not sustainable with increasing the number of concurrent users.

A closer examination of the results reveals that the response times for Fission have a significant number of outliers as the concurrency of requests increases more than one user (i.e., \( n = 1 \)). All frameworks have consistent response time except Fission which has several outliers in both functions. This performance degradation might be related to the router component of Fission that forwards all incoming HTTP requests to the appropriate function. This component has a potential to become a bottleneck as the workload demand increases. For other frameworks such as Kubeless and OpenFaaS which are based on native Kubernetes components, there is more performance stability as they utilize the Kubernetes Ingress controller to route requests and balance the load. More results and analysis can be found in [11].

C. Impact of autoscaling

In this section, we evaluate the impact of autoscaling on the response time in all the serverless frameworks. For this experiment, we used both ML models as listed in Table II. We choose to scale functions based on CPU utilization and set the threshold to 50%, which is the trigger for creation of more function replicas in pods. For the autoscaling experiment, all frameworks start with one pod. For the experiment without autoscaling, the setup was one pod per edge device (i.e., 4 pods in total). Please note that we did not set any resource request and limit for pods. All frameworks, except funcX use the Kubernetes Horizontal Pod Autoscaler to perform scaling based on CPU utilization. We use the JMeter tool to send 1000 requests with 1 and 5 concurrent users and repeat each experiment 3 times. Given the resource limitations we could not execute more concurrency for all the frameworks.

Fig. 4 and Fig. 5 show the results for experiments for all the frameworks for 1 and 5 concurrent users with autoscaling for CNN and LSTM functions, respectively. In these figures, solid lines represent results of autoscaling and dashed lines show the same configuration without autoscaling while colors differentiate between different frameworks. These results show that for CNN function, Kubeless achieved the best response time with and without autoscaling. However, Fission is the only framework that can leverage the autoscaling to improve the response time by 27% and 100% for \( n = 1 \) and \( n = 5 \), respectively. For LSTM function, Fission and funcX can provide a better response time which is due to their ability for more efficient autoscaling as presented in Fig. 5. Fission can leverage the autoscaling to improve the response time by 5% and 30% for \( n = 1 \) and \( n = 5 \), respectively.

We also examine the variation of pod numbers during a single iteration of the experiment for both CNN and LSTM functions as plotted in Fig. 6 and Fig. 7. As it can be seen in these figures, Fission and Kubeless are adding more pods in response to an increase in the CPU utilization due to higher workload. OpenFaaS handle autoscaling with a more conservative approach and add less pods compared to the other two frameworks. As mentioned earlier, funcX does not support autoscaling with CPU utilization, so there is not much
Fig. 4: CDF of response time for CNN function (solid lines: autoscaling, dashed lines: without autoscaling)

Fig. 5: CDF of response time for LSTM function (solid lines: autoscaling, dashed lines: without autoscaling)

Fig. 6: Number of pods for autoscaling of CNN function for different frameworks

Fig. 7: Number of pods for autoscaling of LSTM function for different frameworks
change in the number of pods. We also notice that the total duration of the experiment is longer for OpenFaaS due to the higher average response time where all 1000 requests need to be processed. This is justified by the fact that each JMeter’s thread is executed in a synchronized fashion and waits for a response before sending the next request.

In addition, we examine the ratio of successfully received responses (i.e., success rate) under different levels of concurrent requests for both machine learning workloads. For one user, both Kubeless and Fission show a reliable behaviour with 100% success rate, but OpenFaaS and funcX have 1-2% of error. This pattern is repeated for higher concurrency levels, but we observed less success rate for funcX in this case with an error up to 3%.

| Machine learning model | Training time (sec) x86 | ARM | Accuracy |
|-------------------------|--------------------------|-----|----------|
| CNN                     | 455                      | 6790| 0.99     |
| LSTM                    | 163                      | 1203| 0.90     |

**TABLE III: Machine learning training results**

**D. Model training**

In this section, we consider the performance of edge platforms for machine learning training. As stated in section III-A, for the machine learning workloads we trained the models on a x86 machine and execute the model inference as the serverless function. As a new experiment, we run the model training on a Raspberry Pi with an ARM processor and reported the results in Table III. It is noted that, the CNN model has a higher accuracy, but it takes a longer amount of time to train over both processor architectures. However, shorter run times are observed for CNN classification on the edge compared to LSTM as illustrated in Fig. 4 and Fig. 5. A slower execution time for classification observed for the LSTM model in part relates to its higher cell structure complexity and its sequential execution model, which is less amenable to parallel processing compared to a CNN network architecture.

Although the CNN model is clearly a more preferable choice in terms of accuracy, implementing two different deep learning network architectures highlights their significantly different behaviours, such as during pod recruitment in autotasking, that may favour one architecture over another. These algorithmic considerations become very relevant during edge processing conditions, where resource limitations exacerbate small differences in low level container behaviour, in comparison to a less constrained CPU processing environment. The choice of edge-based machine learning models needs to be carefully considered for both classification and training modes, but more so in the latter case due to the significantly heavier computational demands involved. As derived from Table III, the ARM/x86 training time ratio of 7.38 for LSTM compared to 15.58 for CNN demonstrates the added processing overhead that can potentially be saved by simply choosing a more suitable machine learning model for edge training. A compromise may need to be made between choosing an efficient model that is optimised for training on the edge, despite resulting in an overall decreased classification accuracy.

**IV. CONCLUSIONS AND FUTURE WORK**

In this paper, the performance evaluation of serverless edge computing under different machine learning workloads has been analyzed. The selected test-bed is based on the most popular serverless frameworks with distinct features, including Kubeless, OpenFaaS, Fission and funcX. We found that Kubeless outperforms the other frameworks in terms of response time for basic workloads, and Fission has the worst performance for handling concurrent users. In addition, Node.js is a considerably faster language compared to Python and can be considered to be the selected deployment language for real-time functions. For machine learning workloads, we demonstrated that based on the structure and complexity of the model, frameworks such as Fission could have a better performance with limited concurrent users. Moreover, if we do not set any resource limit on pods, it might be better not to use autoscaling in some cases as it has some performance degradation for frameworks such as OpenFaaS. In future work, we intend to work on modifying some of the critical components of serverless frameworks, including the function handling and autoscaling modules, to improve their performance for edge computing platforms.

**REFERENCES**

[1] F. Samie, L. Bauer, and J. Henkel, “From cloud down to things: An overview of machine learning in internet of things,” *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4921–4934, 2019.

[2] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz, “The cloud is not enough: Saving IoT from the cloud,” in 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15), 2015.

[3] M. Satyanarayanan, “The emergence of edge computing,” *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[4] F. Mehdipour, B. Javadi, A. Mahanti, and G. Ramirez-Prado, “Fog Computing Realization for Big Data Analytics,” *Fog and Edge Computing: Principles and Paradigms*, pp. 259–290, 2019.

[5] 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.

[6] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar et al., “Cloud programming simplified: A berkeley view on serverless computing,” arXiv preprint arXiv:1902.03383, 2019.

[7] S. Eismann, J. Scheuener, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup, “The state of serverless applications: Collection, characterization, and community consensus,” *IEEE Transactions on Software Engineering*, 2021.

[8] B. Javadi, J. Sun, and R. Ranjan, “Serverless architecture for edge computing,” in *Edge Computing: Models, technologies and applications*. Institution of Engineering and Technology, 2020, pp. 249–264.

[9] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sharski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, “Serverless edge computing: vision and challenges,” in *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.

[10] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, and T. Huang, “When serverless computing meets edge computing: Architecture, challenges, and open issues,” *IEEE Wireless Communications*, vol. 28, no. 5, pp. 126–133, 2021.

[11] Q. L. Trieu, B. Javadi, J. Basilakis, and A. N. Toosi, “Performance evaluation of serverless edge computing for machine learning applications,” arXiv preprint arXiv:2210.10331, 2022.