An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge

An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge

Andrei Palade, Aqeel Kazmi and Siobhán Clarke
School of Computer Science and Statistics, Trinity College Dublin, Ireland,
{apalade,aqeel.kazmi,siobhan.clarke}@scss.tcd.ie

***Abstract*—The proliferation of Internet of Things (IoT) and the success of resource-rich cloud services have pushed the data processing horizon towards the edge of the network. This has the potential to address bandwidth costs, and latency, availability and data privacy concerns. Serverless computing, a cloud computing model for stateless and event-driven applications, promises to further improve Quality of Service (QoS) by eliminating the burden of always-on infrastructure through ephemeral containers. Open source serverless frameworks have been introduced to avoid the vendor lock-in and computation restrictions of public cloud platforms and to bring the power of serverless computing to on-premises deployments. In an IoT environment, these frameworks can leverage the computational capabilities of devices in the local network to further improve QoS of applications delivered to the user. However, these frameworks have not been evaluated in a resource-constrained, edge computing environment. In this work we evaluate four open source serverless frameworks, namely, Kubeless, Apache OpenWhisk, OpenFaaS, Knative. Each framework is installed on a bare-metal, single master, Kubernetes cluster. We use the JMeter framework to evaluate the response time, throughput and success rate of functions deployed using these frameworks under different workloads. The evaluation results are presented and open research opportunities are discussed.**

I. Introduction

Next generation technologies such as self-driving cars, smart cities or augmented reality services require new approaches to deal with the network traffic generated by the IoT devices deployed to enable such technologies [1]–[5]. In this context, edge computing has emerged as a promising solution to satisfy the Quality of Service (QoS) requirements of such applications by pushing the data processing horizon towards the edge of the network, and by relieving devices from computationally-intensive tasks, to reduce energy consumption [6]–[9]. Serverless computing, a well-known cloud computing paradigm also sometimes referred to as Function-as-a-Service (FaaS) has eliminated the need for always-on infrastructure through ephemeral containers [10]. These containers can be stopped and destroyed, then rebuilt and replaced with an absolute minimum set up and configuration. This event-driven service execution model enables on-demand access to functions (or services), which has the potential to address bandwidth costs, and latency, availability and data privacy concerns introduced by the IoT devices [11]. Incorporating serverless computing at the edge of an IoT network for executing small tasks may reduce the overall processing time of these tasks [12], [13].

Currently, all major cloud service providers offer serverless computing platforms (e.g., AWS Lambda [14], Azure Functions [15], IBM Cloud Functions [16], and Cloud Functions [17]). However, such platforms require functions to be written or deployed in a certain way, which results in vendor lock-in [18]. Several open-source FaaS frameworks have been proposed to allow to run serverless computing on private infrastructure, thereby avoiding any forms of vendor lock-in. Recent studies such as Mohanty et al. [18] and Kritikos et al. [19] have evaluated the usefulness and performance of selected open source serverless frameworks, but such studies have not considered the constraints introduced by an edge-based environment.

This paper presents an evaluation of four open source serverless frameworks in an edge computing environment. The evaluated frameworks include Kubeless [20], Apache OpenWhisk [21], OpenFaaS [22], and Knative [23]. The experimental setup consists of services deployed on IoT devices that constantly push measured data about environmental parameters to a serverless-runtime deployed on edge-devices located in the same local network as the IoT devices. The evaluation focuses on qualitatively and quantitatively measuring the performance of serverless frameworks deployed in an edge environment. The evaluation captures the innovative capabilities introduced by bringing such technologies at the edge of the network to be used for processing data generated by IoT devices.

The paper is organised as follows: Section II outlines the requirements of an open-source serverless computing framework through an IoT scenario. Section III summarises the selection methodology and presents an overview of four representative frameworks. Section IV introduces the experimental setup and the metrics used in the evaluation. Section V presents the results of the evaluation. Open research challenges including possible future research directions are presented in Section VI. Section VII outlines related work. Finally, Section VIII concludes the work and points to areas of potential future work.

II. Use Case Scenarios and Requirements

Measuring human vital signs in disasters or in every day life are common use cases that require low latency [12]. In case of a major disaster, prompt paramedic attention is necessary to save people's lives. User wearable sensors can provide critical information about patient's medical condition and help determine a priority queue for patient monitoring. In case of every day life monitoring, sensors can continuously stream

TABLE I
GITHUB CONTRIBUTORS AND FOLLOWERS AS OF APRIL 18TH, 2019.

Name	Contributors	Stars	Reference
Apache Openwhisk	151	3955	[21]
Knative	117	1655	[23]
OpenFaaS	99	13915	[22]
Kubeless	76	4523	[20]
Fission	76	4261	[26]
Fn	76	3940	[27]
Nuclio	36	2655	[28]
Iron Functions	32	2568	[29]
OpenLambda	17	594	[30]

data of electrocardiogram readings to a nearby edge device to perform data analytics tasks. In such scenarios, a cloud-based approach for such data processing tasks may not always be feasible because of:

- **High Latency.** Moving a large amount of data to cloud may be more expensive than processing it locally at the edge.
- **Privacy Concerns.** Some tasks may contain confidential data, which makes it infeasible to transfer and process data in the cloud.
- **Mobility Support.** Non-stationary sensing devices may introduce frequent disconnections, which increases the resolution time and reduces the availability of applications developed for IoT environments.

One of the main drivers of edge computing is low latency support. In this context, a serverless computing framework can perform the operational procedures of the server, network, load balancing and scaling. A function should be launched instantaneously in response to an event [24]. A serverless computing framework should provide the ability for auto-scaling (or scale to zero) to minimise or to avoid the resource usage of running the serverless-runtime (running & idle). The framework may run zero to thousands of instances of the function. This is based on demand for that function. A serverless framework should handle load spikes, and provide resource quotas. Also, the scaling of the functions should be performed without knowledge of the application. As functions run arbitrarily code from multiple IoT devices, they must remain well isolated from the host platform while they are required to satisfy certain QoS requirements [25].

Other concerns that should be considered are statefulness (state management in stateless functions), security (when running multiple functions on a shared platform), support for legacy applications and cross-cloud support. These concerns are not discussed here due to space limitations as we also believe that such issues should be addressed in a separate work.

III. METHODOLOGY FOR FRAMEWORK SELECTION

Recently, a number of open-source serverless frameworks have been proposed such as Kubeless [20], Apache OpenWhisk [21], OpenFaaS [22], Knative [23], Fn [27], Iron Functions [29], Nuclio [28], Fission [26], and OpenLambda [30]. An important criterion when selecting an open-source project is the strong developer community around that project. In this work, as selection criterion, the number of contributors and followers of the source-code repository associated with each framework is used. Table I outlines the number Github contributors and the number of users starring the repository. The number of Github contributors is considered first, and, in case of a tie, the number of Github stars is used. Using this criterion, Apache Openwhisk, Knative, OpenFaaS, Kubeless and Fission have the highest Github number of contributors. The number of Github stars is used as a tie-breaker for Kubeless and Fission.

Kubeless. Kubeless is a Kubernetes-native serverless framework. The Kubeless programming model is based on three primitives: functions, triggers and runtime. A function is a representation of the code to be executed, and trigger is an event source. A trigger can be associated to a single function or to a group of functions depending on the event source type. Kubeless ensures that the associated function(s) are invoked at least once. A runtime represents a language and runtime specific environment in which a function will be executed. Kubeless uses Custom Resource Definitions (CRDs) to extend Kubernetes API, which allows developers to interact with functions as if they were native Kubernetes objects.

The main component of this platform is a CRD controller that continuously watches for changes to function objects and takes the necessary actions, such as creation or deletion of a new function object. The runtime image used to deploy a function can be explicitly specified by the user, the image artifact is generate on-the-fly, or a pre-built image is used where using Kubernetes' configmap the function code is deployed into the corresponding Kubernetes pod. Upon deletion, the controller releases the used computing resources.

Apache OpenWhisk. The OpenWhisk programming model is based on three primitives: actions, triggers and rules. An action is a stateless function that executes code, and a trigger is a class of events that can originate from various sources. A rule associates a trigger to an action. Multiple actions from different languages may be composed together to create a longer processing pipeline called a sequence. The polyglot nature of the composition process decouples the orchestration of the dataflow between functions from the choice of language.

The main components of this platform are: an Nginx webserver, a controller component, an Apache Kafka component, an Invoker component, and a CouchDB database for storing the user credentials, action metadata, namespaces, and the definitions of actions, triggers, and rules. The Nginx webserver is used as a reverse proxy for the entire system. The controller component performs the authentication, authorisation and routing of every request before handing over the control to the next component. The Kafka component is used to manage the connection between the controller and Invokers. The Invoker component copies the code from the CouchDB component and injects that into a Docker container. Also this component maintains the list of active Docker containers where actions are deployed. When the execution

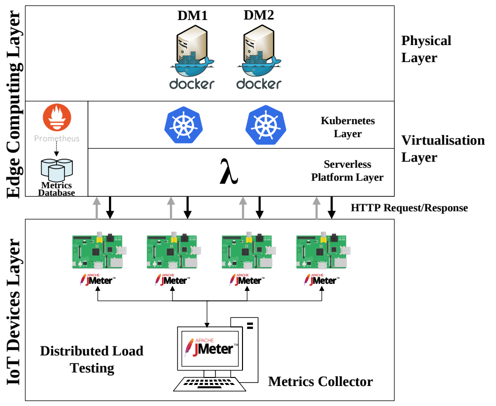

Fig. 1. Deployment Setup.

of a certain action is finalised, the result is stored in the CouchDB component for retrieval.

OpenFaaS. The OpenFaaS programming model is based on one primitive: functions. The developer needs to provide a handler and a function. The main component of this platform is: an API gateway. The API gateway provides access to the functions, collects metrics and provides scaling by interacting with the orchestration engine (i.e., Kubernetes). A command line interface is used to package each function into a Docker container. Each container contains a *watchdog* (i.e., a webserver that acts as an entry point to the container and invokes the function). OpenFaaS uses the AlertManager component (combined with Prometheus) or the Kubernetes Horizontal Pod Scaler (HPA) to enable the zero-scale feature.

Knative. The Knative framework is built on top of Kubernetes [31] and Istio [32], which provide application (container-based) runtime and advanced network routing. This allows Knative to extend Kubernetes platform using CRDs to enable a higher-level of abstractions.

Knative is a set of building blocks for serverless platforms running on top of Kubernetes. The main components of this platform are: Build, Serving and Eventing. The Build component is implemented using a Kubernetes CRD and is a pluggable model for building applications (in containers) from source code. Serving extends Kubernetes to provide runtime computing support for deploying and running serverless workloads. This component provides scale-to-zero support based on the received requests, and it uses Istio for network routing. The Eventing component provides the necessary primitives for consuming and producing events. Knative is not a complete serverless platform and leaves the higher-level API concepts, CLIs, tooling, etc. up to specific vendors to implement (e.g., combining this platform with Apache OpenWhisk [33]).

IV. Experimental Setup

A. Common Experimental Architecture

We deploy each serverless framework on top of a common experimental architecture (Figure 1). This architecture is composed of two layers: an Edge Computing Layer and an IoT Devices Layer. The Edge Computing Layer is divided in two sub-layers: the physical layer and the virtualisation layer. The physical layer contains the physical machines used to support the Edge Computing layer. The physical layer is represented by two Desktop Machines (DM) where first machine (DM1) is equipped with an Intel(R) Core(TM) i7-3770 3.40GHz CPU, and the second machine has an Intel(R) Core(TM) i7-2600 3.40GHz CPU. Each machine has 12GB of DDR3 1333 MHz RAM. Each machine is running Ubuntu 16.04.5 LTS machines (Linux 4.4.0-142-generic). In the virtualisation layer, each machine has Docker v18.09.2 installed. The containers on each machine are managed through the Kubernetes v1.13.3 cluster where DM1 is the manager node and DM2 is a worker node. Flannel v0.11.0 is used to build the overlay network in this cluster. Each serverless framework is installed in the cluster and interacts directly with the Kubernetes cluster manager.

The IoT Devices Layer contains four Raspberry PI devices used as IoT devices: two Raspberry PI 2 Model B v1.1 and two Raspberry PI 3 B v1.2 model. Each device is running JMeter v5.1. Each device will be used 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 a desktop machine that has a JMeter client installed. The specification of this machine is omitted as its only purpose is to display the metrics collected by the JMeter engines running on each IoT device.

B. Evaluation Metrics

1) Qualitative metrics:

- *Open Source License:* freely access to modify, and distribute (in both modified and unmodified form) code to other developers.
- *Developer Community Support:* a framework should have a strong and thriving developer community support. This feature is identified based on the number of code commit frequency, pull requests/merge request frequency, reputation, availability of developer support through extensive documentation, mailing lists or chat rooms.
- *Programming Language Support:* a framework should offer support for multiple languages. Also, it should be possible to add support for other languages.
- *Container Orchestration Engine Support:* a framework should offer support for multiple container orchestration engines (e.g., Kubernetes [31], Docker Swarm [38], Mesos [39], Nomad [40], Kontena [41]) to provide more flexibility for both the developers and operations team. These orchestration engines provide an abstraction layer between the application containers that run on the available resources, and the actual resource pools.
- *Monitoring Support:* a framework should have an integrated monitoring tool that can help the operations team to monitor the performance metrics of a deployed function, such as the number of invocations or the execution time of a function.
- *Function Triggers:* a framework should offer support for both synchronous (HTTP-based) and asynchronous

TABLE II
RESULTS OF THE QUALITATIVE EVALUATION AND THE EASE OF DEPLOYMENT MEASUREMENTS.

	Kubeless	**Apache Openwhisk**	**OpenFaaS**	**Knative**
Open Source License	Apache License 2.0	Apache License 2.0	MIT License	Apache License 2.0
Programming Language Support	Ballerina (v0.981.0), Go (v1.10), Java (v1.8), NodeJS (v6, v8), PHP (v7.2), Python (v2.7, v3.4, v3.6), Ruby (v2.3, v2.4, v2.5), .NET Core (v2.0) [34]	Ballerina (v0.990.2.), Go (v1.11), Java (v1.8), NodeJS (v6, v8, v10), PHP (v7.3), Python (v2.7, v3.6), Ruby (v2.5), Swift (v4.2), .NET Core, C# , Docker actions [35]	Go (v1.10), Java (v1.8), NodeJS (v8.9.1), PHP (v7.2), Python (v2.7, v3.6), Ruby (v2.5.1) C#, Docker file [36]	Go (v1.12), Java (v1.8 or later), NodeJS (v10), Kotlin (v1.2.61), PHP (v7.2), Python (v2.7 or later), Ruby (v2.3 or later), Scala (latest version), .NET Core (v2.1), C# [37]
Container Orchestration Engine Support	Kubernetes	Kubernetes	Kubernetes, Docker Swarm, Apache Mesos	Kubernetes
Monitoring Support (Out-of-the-box)	Prometheus with Grafana	None	Prometheus with Grafana	Prometheus with Grafana
Function Triggers	HTTP and other event sources	HTTP or Feeds triggers	HTTP and other event sources	HTTP or Message Broker
Auto-scaling	Yes	Yes	Yes	Yes
CLI Interface	kubeless	wsk	faas-cli	kubectl
Ease of Deployment (mins)	5	15	10	20

(event-based) triggers.

- *CLI Interface:* the availability of the command line interface, which should ease of the management of functions and allow for better integration with third party tools such as event-driven triggers.

2) Quantitative metrics: Serverless functions are expected to serve infrequent and sporadic demands. The framework must scale to efficiently utilise the available physical infrastructure with varying levels of incoming traffic.

- *Response Time:* the resolution time of the request.
- *Throughput:* the number of satisfied requests (transactions) per second.
- *Success Rate:* the ratio between the number of successful requests and the total number of requests.
- *Ease of Deployment:* a time metric showing the duration from when the initialisation script is triggered until all the components are deployed.

C. Test Case Generation

The JMeter tool is configured to perform 10 requests with various levels of concurrency (1, 5, 10, 20 concurrent requests). The concurrency level affects the number of requests received simultaneously by the framework. We created a NodeJS function that receives HTTP request and replies with a confirmation message. The header of this request includes a value, which represents a temperature reading. This function is installed in each framework. We chose this function to have minimal overhead in terms of business logic and code dependencies. We measure the response time, throughput and the success rate of received responses under various levels of concurrency. An independent replication method is chosen with 50 iterations to achieve adequate statistical significance. We measure the impact of auto-scaling on each evaluated metric. The CPU utilisation is used as a metric to perform the auto-scaling, and is set to 50% in this evaluation. When utilisation exceeds this threshold, the creation of a new function is triggered. All frameworks rely on the Kubernetes' Horizontal Pod Autoscaler to perform scaling based on the CPU utilisation.

V. RESULTS

Table II shows the results of the qualitative evaluation performed in this paper. The results show that each framework has similar features in offering. An evaluation of ease of the deployment is also attached here to show how quickly each framework can be deployed. The values recorded represent the duration from the moment the initialisation command is triggered until all the required components of the framework are deployed. These measurements are performed using the common experimental architecture presented in Section IV-A. While Knative does not have an official CLI interface, third-party implementations such as knctl [42] are available.

Figure 2 shows the results of the quantitative evaluation. While all the frameworks leave the scaling decisions to the Kubernetes HPA feature, we observed that the values obtained for the evaluated metrics vary considerably. For instance, Apache OpenWhisk has the worst performance of all evaluated frameworks for all the metrics. In case of one service/device, its success rate is similar to the other three frameworks but decreases considerably as the load increases. This performance degradation is because of the (centralised) Nginx component, which handles all the received HTTP requests, becomes a bottleneck. While response time and throughput improves as the load increases, a large number of requests made by the services deployed on IoT devices receive 429 Too Many Requests. Further configuration of this component, may reduce the performance bottleneck.

The difference in success rate and throughput for the Kubeless, OpenFaas and Knative is not observable. However, we observe that Kubeless scales better in terms of response time, as the load increases. Kubeless maintains on average 12.57 to 13.79 ms as the number of services per IoT device is increased, whereas the response time increases in OpenFaaS from 96.92 to 106.82 ms and in Knative from 86.27 to 253.66 ms on average.

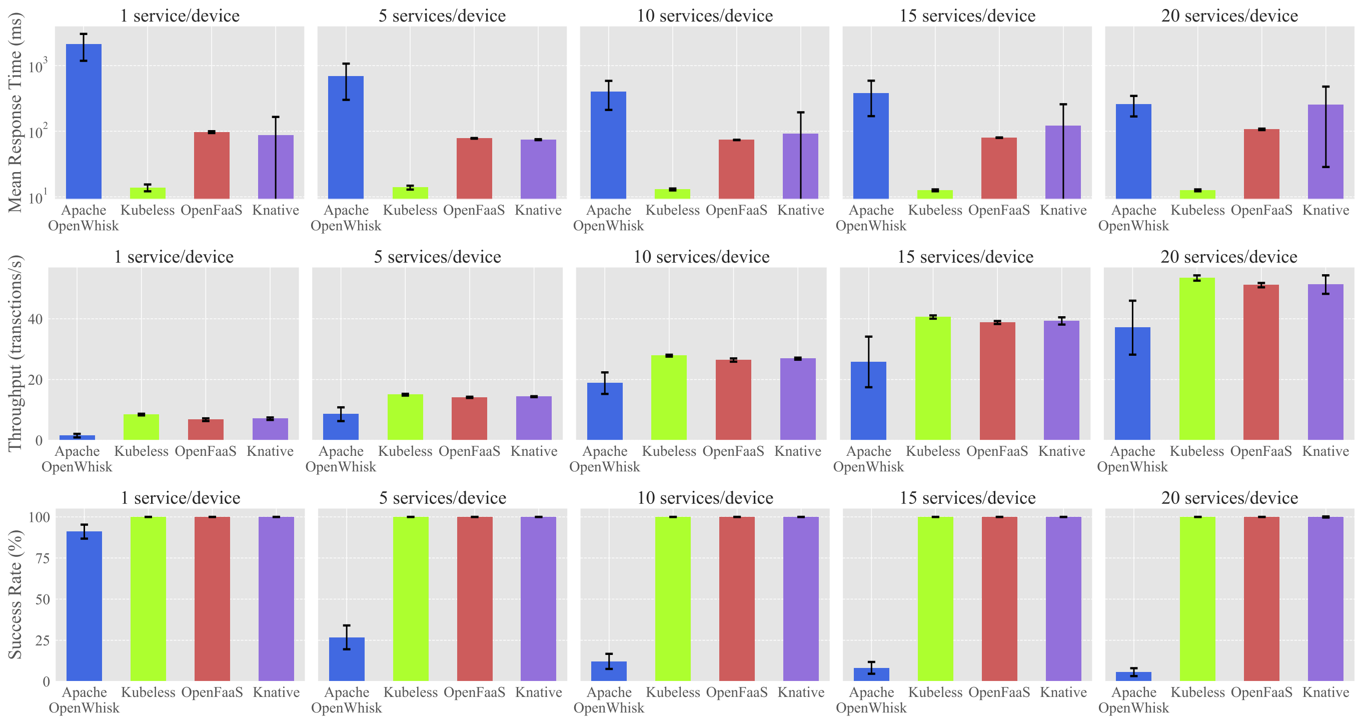

Fig. 2. Results of the Quantitative Evaluation.

VI. Open Research Challenges

In the context of this evaluation we observed that a number of open research challenges introduced by running serverless platforms at the edge of the network:

- **Observability.** This includes monitoring, alerts, log aggregation and distributed system tracing. Achieving log aggregation and distributed system tracing implies more effort, especially when using external services because of lack of agents or deamons monitoring the functions. Open-source projects such Zipkin [43] may be explored.
- **Resource limitations.** Functions have limitations in regards to memory allocation, timeout, payload sizes, deployment sizes, concurrent executions, etc. Such boundaries should be consistent with the serverless philosophy where a small function with a single responsibility should run in a short time with low memory allocation. To what extent of these technologies can be used for long running processes with high memory footprint should be explored.
- **Lack of QoS Support.** Users have little or no control over the QoS of functions deployed using such frameworks. The auto-scaling feature does not provide any QoS guarantees. These frameworks should consider the user's or provider's objectives in a coordinated manner.
- **Fault Tolerance.** The evaluated frameworks have limited support for fault tolerance. In case of a failed container, a basic retry-mechanism is used. An open research challenge is to explore the existing fault detection mechanisms employed and scheduling functions on other edge nodes [44].
- **Function Composition.** While FaaS functions enable users to quickly deploy small services, any more complex use cases require multiple functions. Although existing tools to develop functions' chains have been developed (e.g., Fission worksflows [26]), how to efficiently and effectively perform function composition and placement remains to be explored especially in an edge environment.

VII. Related Work

Serverless computing has received a significant amount of attention because of the potential benefits in configuration and management overhead reduction. Baldini et al. [45] highlighted this in a survey of several serverless frameworks. In their study, the authors outlined use cases for such frameworks, and identified some open technical challenges to enable the serverless computing vision. Lynn et al. presented a multi-level feature analysis and feature comparison of seven enterprise serverless computing platforms [46].

Open-source serverless frameworks have been introduced to avoid the vendor lock-in. Mohanty et al. [18] performed a feature comparison of four open source serverless frameworks (Kubless, OpenFaaS, Fission, and OpenWhisk). The authors also evaluated the performance of three frameworks (excluding OpenWhisk) when deployed on Kubernetes cluster. In another study, Kritikos et al. [19] reviewed and provided a feature analysis comparison of seven open source serverless frameworks and outlined a set of challenges that require attention from research community. These works have not considered open source frameworks in the context of edge computing.

A number of works have investigated using the serverless computing technology at the edge for executing data processing functions. These works have identified the challenge of auto-scaling, which is not predictable to the user. When a function is scaled down, the *cold start* problem can cause latency issues in a constrained environment [10], [47]–[49].

Nastic et al. [12] presented a serverless real-time data analytics platform to support data processing at the edge. Cicconetti et al. [50] propose a fully distributed delegation architecture to address the limitations of existing serverless platforms that require a logically centralised controller for task scheduling. While Kuntsevich et al. [51] proposed a method to benchmark the Openwhisk platform, a comparison of this with existing open-source solutions support in an edge environment have not been previously presented.

VIII. Conclusion and Future Work

The traditional way of deploying IoT applications is to use IoT infrastructure in conjunction with cloud resources to perform data processing. However, this technique adds high latency even for small tasks. Incorporating serverless computing at the edge of an IoT network for executing small tasks can reduce the overall processing time of these tasks. To demonstrate the viability of such an approach, this article has quantitatively and qualitatively evaluated four open-source serverless computing frameworks in an edge environment. We presented some typical scenarios where such platforms may be used. We also presented a set of requirements that a serverless computing framework may need to provide to enable the potential features of serverless computing at the edge of the network. We found that Kubeless outperforms the other frameworks across the proposed scenarios in terms of response time and throughput. Apache OpenWhisk has the worst performance. Future work will explore the support for composition of functions offered by these frameworks.

References

[1] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for Internet of Things: a Survey," *IEEE Internet of things journal*, vol. 3, no. 1, 2016.

[2] C. Cabrera, A. Palade, and S. Clarke, "An Evaluation of Service Discovery Protocols in the Internet of Things," in *SAC*. ACM, 2017.

[3] A. Palade, C. Cabrera, G. White, M. A. Razzaque, and S. Clarke, "Middleware for Internet of Things: A Quantitative Evaluation in Small Scale," in *2017 IEEE 18th WoWMoM*. IEEE, 2017.

[4] C. Cabrera, G. White, A. Palade, and S. Clarke, "The Right Service at the Right Place: a Service Model for Smart Cities," in *2018 IEEE PerCom*. IEEE, 2018.

[5] C. Cabrera, A. Palade, G. White, and S. Clarke, "Services in IoT: A Service Planning Model Based on Consumer Feedback," in *International Conference on Service-Oriented Computing*. Springer, 2018.

[6] G. White, A. Palade, and S. Clarke, "Qos Prediction for Reliable Service Composition in IoT," in *ICSOC*. Springer, 2017.

[7] A. Palade, C. Cabrera, G. White, and S. Clarke, "Stigmergic Service composition and Adaptation in Mobile Environments," in *International Conference on Service-Oriented Computing*. Springer, 2018.

[8] A. Palade and S. Clarke, "Stigmergy-Based QoS Optimisation for Flexible Service Composition in Mobile Communities," in *2018 IEEE World Congress on Services (SERVICES)*. IEEE, 2018.

[9] G. White, A. Palade, C. Cabrera, and S. Clarke, "IoTPredict: Collaborative QoS Prediction in IoT," in *2018 IEEE IPerCom*. IEEE, 2018.

[10] D. Pinto, J. P. Dias, and H. Sereno Ferreira, "Dynamic Allocation of Serverless Functions in IoT Environments," in *IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, 2018.

[11] A. Hall and U. Ramachandran, "An Execution Model for Serverless Functions at the Edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*. ACM, 2019.

[12] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A Serverless Real-Time Data Analytics Platform for Edge Computing," *IEEE Internet Computing*, vol. 21, no. 4, 2017.

[13] G. White, C. Cabrera, A. Palade, and S. Clarke, "Augmented Reality in IoT," in *ICSOC*, 2018.

[14] "AWS Lambda," https://aws.amazon.com/lambda/, online: 2019-01-20.

[15] "Azure Functions," https://azure.microsoft.com/en-us/services/functions/, online: 2019-01-20.

[16] "IBM Cloud Functions," https://www.ibm.com/cloud/functions/, online: 2019-01-20.

[17] "Google Cloud Functions," https://cloud.google.com/functions/, online: 2019-01-20.

[18] S. K. Mohanty, G. Premsankar, and M. Di Francesco, "An Evaluation of Open Source Serverless Computing Frameworks," in *2018 IEEE CloudCom*. IEEE, 2018.

[19] K. Kritikos and P. Skrzypek, "A Review of Serverless Frameworks," in *2018 IEEE/ACM UCC Companion*. IEEE, 2018.

[20] "Kubeless," https://kubeless.io/, online: 2019-03-20.

[21] "OpenWhisk," https://openwhisk.apache.org/, online: 2019-03-20.

[22] "OpenFaaS," https://www.openfaas.com/, online: 2019-03-20.

[23] "Knative," https://github.com/knative/, online: 2019-03-20.

[24] G. Premsankar, M. Di Francesco, and T. Taleb, "Edge Computing for the Internet of Things: A Case Study," *IEEE Internet of Things Journal*, vol. 5, no. 2, 2018.

[25] R. Koller and D. Williams, "Will Serverless End the Dominance of Linux in the Cloud?" in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. ACM, 2017.

[26] "Fission," https://fission.io/, online: 2019-03-12.

[27] "Fn Project," https://fnproject.io/, online: 2019-03-12.

[28] "Nuclio," https://nuclio.io/, online: 2019-03-12.

[29] "Iron Functions," https://open.iron.io/, online: 2019-03-12.

[30] "OpenLamda," http://www.open-lambda.org, online: 2019-04-16.

[31] "Kubernetes," https://kubernetes.io/, online: 2019-01-20.

[32] "Istio," https://istio.io/, online: 2019-01-20.

[33] "Knative," https://cwiki.apache.org/confluence/display/OPENWHISK/%5BWIP%5D+OpenWhisk+on+Knative, online: 2019-03-20.

[34] "Kubeless," https://kubeless.io/docs/runtimes/, online: 2019-03-01.

[35] "OpenWhisk Runtime," https://github.com/apache/incubator-openwhisk/blob/master/docs/actions.md, online: 2019-02-26.

[36] "OpenFaaS," https://github.com/openfaas/templates, online: 2019-02-27.

[37] "Knative Runtime Support," https://github.com/knative/docs/tree/master/serving/samples, online: 2019-03-01.

[38] "Docker," https://docs.docker.com/engine/swarm/, online: 2019-01-20.

[39] "Apache Mesos," http://mesos.apache.org/, online: 2019-01-20.

[40] "Nomad," https://www.nomadproject.io/, online: 2019-01-20.

[41] "Kontena," https://www.kontena.io/, online: 2019-01-20.

[42] "knctl," https://github.com/cppforlife/knctl, online: 2019-04-18.

[43] "Zipkin," https://github.com/apache/incubator-zipkin, online: 2019-04.

[44] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless Edge Computing: Extending Serverless Computing to the Edge of the Network," in *SYSTOR*, 2017.

[45] I. Baldini, P. C. Castro, K. S.-P. 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.

[46] 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 CloudCom*, Dec 2017.

[47] E. d. Lara, C. S. Gomes, S. Langridge, S. H. Mortazavi, and M. Roodi, "Hierarchical Serverless Computing for the Mobile Edge," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[48] L. F. Herrera-Quintero, J. C. Vega-Alfonso, K. B. A. Banse, and E. Carrillo Zambrano, "Smart ITS Sensor for the Transportation Planning Based on IoT Approaches Using Serverless and Microservices Architecture," *IEEE Intelligent Transportation Systems Magazine*, 2018.

[49] J. Franz, T. Nagasuri, A. Wartman, A. V. Ventrella, and F. Esposito, "Reunifying Families after a Disaster via Serverless Computing and Raspberry Pis," in *2018 IEEE LANMAN*, 2018.

[50] C. Cicconetti, M. Conti, and A. Passarella, "An Architectural Framework for Serverless Edge Computing: Design and Emulation Tools," in *2018 IEEE CloudCom*. IEEE, 2018.

[51] A. Kuntsevich, P. Nasirifard, and H.-A. Jacobsen, "A Distributed Analysis and Benchmarking Framework for Apache OpenWhisk Serverless Platform," in *Middleware Conference*. ACM, 2018.