AN EVALUATION OF OPEN SOURCE SERVERLESS CLOUD COMPUTING FRAMEWORKS

PAMBALANAGESWARARAO 1, J RANJITH 2, EEDUNURI MURALIDHAR REDDY 3
4.THAMBI VINOD KUMAR, 5.K.VENKATA RAMANA
Assistant Professor, Department of Computer Engineering, Ellenki college of Engineering and Technonlogy, patelguda (vi)' near BHEL ameenpur (m), Sangareddy Dist. Telangana 502319.

Abstract

Recent advancements in virtualization and software architecture have led to the new paradigm of serverlesscomput- ing, which allows developers to deploy applications as stateless functions without worrying about the underlying infrastructure. Accordingly, a serverless platform handles the lifecycle, exe- cution and scaling of the actual functions; these need to run only when invoked or triggered by an event. Thus, the major benefits of serverless computing are low operational concerns and efficient resource management and utilization. Serverless computing is currently offered by several public cloud service providers. However, there are certain limitations on the public cloud platforms, such as vendor lock-in and restrictions on the computation of the functions. Open source serverless frameworks are a promising solution to avoid these limitations and bring the power of serverless computing to on-premise deployments. However, these frameworks have not been evaluated before. Thus, we carry out a comprehensive feature comparison of popular open source serverless computing frameworks. We then evaluate the performance of selected frameworks: Fission, Kubeless and OpenFaaS. Specifically, we characterize the response time and ratio of successfully received responses under different loads and provide insights into the design choices of each framework.

Index Terms—serverless computing, function-as-a-service, Kubeless, Fission, OpenFaaS, performance evaluation

I. INTRODUCTION

Serverless computing is an emerging paradigm wherein soft- ware applications are decomposed into multiple independent stateless functions [1, 2]. Functions are only executed in re- sponse to triggers (such as user interactions, messaging events or database changes), and can be scaled independently as they are completely stateless. Hence, serverless computing is also sometimes referred to as function-as-a-service (FaaS) [3]. In this approach, almost all operating concerns are abstracted away from developers. In fact, developers simply write code and deploy their functions on a serverless platform [4]. The platform then takes care of function execution, storage, con- tainer infrastructure, networking, and fault tolerance. Addition- ally, the serverless platform takes care of scaling the functions according to the actual demand. Serverless computing has been identified as a promising approach for several applications, such as those for data analytics at the network edge [5, 6], scientific computing [7] and mobile computing [8].

In serverless computing, the infrastructure is generally man- aged by a third-party service provider or an operations team when using a private cloud. Currently, all major cloud service providers offer solutions for serverless computing, namely, Amazon Web Services (AWS) Lambda, Azure Functions, IBM

Cloud Functions and Google Cloud Functions. However, these platforms require the functions to be written in a certain way, resulting in

vendor lock-in [2, 9]. Moreover, developers have to rely on the serverless provider's release cycle and additional services from the cloud platform such as message queuing and data storage. They also have to comply with constraints on function code size, execution duration and concurrency [10]. Open source FaaS frameworks are a promising solution to bring the power of serverless computing on-premise. Such frameworks provide more flexibility (for deploying applica- tions, configuring the framework, etc.) and thereby avoid vendor lock-in. For instance, open source frameworks can be deployed both on the edge/fog devices as well as on the public cloud for distributed data analytics [5, 6]. In this regard, a serverless framework should be easy to set up, configure and manage; it should also provide certain performance guarantees. Although recent works have focused on serverless platforms in the public cloud [10, 11], none has evaluated open source FaaS frameworks. In contrast, this work provides a comprehensive feature comparison of popular open source serverless frame- works, namely, Kubeless [12], OpenFaaS [13], Fission [14] and Apache OpenWhisk [15]. Furthermore, it evaluates the performance (in terms of response time and ratio of successful responses) of these frameworks under different workloads and provides insights into the design choices behind them. It finally examines the impact of auto scaling on performance.

The rest of the article is organized as follows. Section II de- scribes the considered frameworks and analyzes their features. Section III evaluates the performance of selected frameworks. Section IV reviews the related work. Finally, Section V pro- vides concluding remarks as well directions for future work.

II. OPEN SOURCE SERVERLESS COMPUTING FRAMEWORKS

This section describes four popular open source serverless frameworks, namely, Fission, Kubeless, OpenFaaS and Open- Whisk. We chose frameworks with at least 3,000 GitHub stars (a mark of appreciation from users). Table I summarizes their features. All the considered frameworks run each serverless function in a separate Docker container to provide isolation. OpenFaaS, Kubeless and Fission utilize a container orchestra- tor to manage the networking and lifecycle of the containers, whereas OpenWhisk may be deployed with or without an orchestrator. We present a short summary of the frameworks, highlighting their main components.

Fission is an open source serverless computing framework built on top of Kubernetes and using many Kubernetes-nativeconcepts [14]. The framework executes a function inside an environment that contains a webserver and a dynamic language-specific loader required to run the function [18]. An executor controls how function pods are created and scaled. One of the main advantages of Fission is that it can be configured to run a pool of "warm" containers so that requests are served with very low latencies [41].

Kubeless is a Kubernetes-native serverless framework [12]. It uses Custom Resource Definitions (CRDs) [42] to extend the Kubernetes API and create functions as custom objects. This allows developers to use the native Kubernetes APIs to interact with the functions as if they were native Kubernetes objects. The language runtime is packaged in a container image. The Kubeless controller continuously watches for changes to function objects and takes necessary action. For instance, if a function object is created, the controller creates a pod for the function and cleans up resources when the function object is deleted. A function's runtime is encapsulated in a container image and Kubernetes configmaps1 are used to inject a function's code in the runtime.

OpenFaaS is an open source serverless framework for Docker and Kubernetes [13]. The OpenFaaS CLI is used to develop and deploy functions to OpenFaaS. Only the functionand handler has to be supplied by the developer, and the CLI handles the packaging of the function into a Docker container. The container comprises a function watchdog, i.e., a webserver that acts as an entry point for function calls within the framework. An API gateway provides an external interface to the functions, collects metrics and handles scaling by interacting with the container orchestrator

ISSN (PRINT): 2393-8374, (ONLINE): 2394-0697, VOLUME-4, ISSUE-1, 2017
DOI:10.21276/ijcesr.2017.4.1.2

plugin. OpenWhisk is an open source,serverless computing frame- work initially developed by IBM and later part of the Apache Incubator project [15]. It is also the underlying technology of the Cloud Functions FaaSproduct on IBM's public cloud. The OpenWhisk programming model is based on three primitives: Action, Trigger and Rule [8]. Actions are stateless functions that execute code. Triggers are a class of events that can originate from different sources. Rules associate a trigger with an action. The scalability of functions is directly managed by the OpenWhisk controller.

Feature	Kubeless	OpenWhisk	Fission	OpenFaaS
Opensourcelicense	Apache2.0[12]	Apache2.0[15]	Apache2.0[14]	MIT[13]
Frameworkdevelopmentlanguage	Go	Scala	Go	Go
Programminglanguagessupported	Python,Node.js,Ruby,PHP,Go,Java,.NETandcustomcontainers[16]	Javascript, Swift,Python,PHP,Java,Linux binaries (includingGo)andcustomcontainers[17]	Python,Node.js,Ruby,Perl, Go,Bash,.NET,PHPandcustomcon-tainers[18]	Python, C#, Go, Node.js,Rubyandcustomcon-tainers[19]
Autoscalingmetric	CPUutilization,QPSandcustommetrics[20]	QPS	CPUutilization[21]	CPU,QPSandcustommetrics[22]
Containerorchestrator	Kubernetes	No orchestrator required,Kubernetessupported[23]	Kubernetes	KubernetesDocker [24], [25], Swarm other extendable toorchestrators[26]
Functiontriggers	http,event,schedule[27]	http[28],event [29], schedule[30]	http,event,schedule[31]	http[32],event[33]
Messagequeueintegration	Kafka,NATS[34]	Kafka[35]	NATS, stor- Azure agequeue[31]	NATS[33],Kafka[36]
Recommendedmonitoringtool	Prometheus[37]	statsd	Istio[38]	Prometheus[39]
CLIsupport	Yes	Yes	Yes	Yes
Industrysupport	Bitnami	IBM,Adobe,RedHat,Apache Software Foun-dationamongothers	Platform9	VMWare[40]
GitHubstars	3,009[12]	3,303[15]	3,412[14]	10,608[13]
GitHubforks	273[12]	629[15]	277[14]	767[13]
GitHubcontributors	61[12]	120[15]	65[14]	68[13]

TABLE I: Overview of features

III. EVALUATION

We evaluate the performance of Fission, Kubeless and OpenFaaS when deployed on a Kubernetes cluster. We choose Kubernetes as it is the only orchestrator supported by all the considered frameworks. We do not include OpenWhisk due to issues faced in setup and its minimal dependence on Kubernetes for orchestration tasks (the interested reader may refer to [43] for a performance evaluation of OpenWhisk).

A. Experimental setup

We run the experiments on Google Kubernetes En- gine (GKE)2. The deployment on GKE is similar to one on a custom Kubernetes cluster deployed on virtualized instances. The serverless framework interacts directly with the Kuber- netes cluster manager. We use Kubernetes version 1.10.4- gke.2 (the latest version available at the time of writing) to set up a cluster with three worker nodes. Each worker node has 2 vCPUs, 7.5 GB RAM and runs the Container- Optimized OS. The cluster is setup in the europe-north1 region and all nodes are located within the same zone to minimize the network latency they experience.

ISSN (PRINT): 2393-8374, (ONLINE): 2394-0697, VOLUME-4, ISSUE-1, 2017
DOI:10.21276/ijcesr.2017.4.1.2

Unless otherwise stated, we deployed each framework with the default settings in the respective installation guide. We set up Fission version0.8.0 and use the newdeploy executor [41] as it supports auto scaling of functions. We use Kubeless version 1.0.0-alpha.6 and the Nginx Ingress controller to provide routing to the functions. The OpenFaaS installation consists of the following components: gateway (v0.7.9), faas-netes (v0.5.1), Prometheus (v2.2.0), alert manager (v0.15.0-rc.0), queue worker (v0.4.3) and faas-cli (v0.6.9). The HTTP watchdog mode is used as this will be the default mode of OpenFaaS in the future.

We use the Apache Benchmark (ab) tool3 to generate HTTP requests that invoke the functions deployed on each frame- work. We run the ab tool on a virtual machine (VM) located in the same zone as the Kubernetes cluster, again, to minimize network latency. The VM has 2 vCPUs, 7.5 GB RAM and runs Debian GNU/Linux 9.4 (stretch) OS. We configure the ab tool to send 10,000 requests with different levels of concurrency (1, 5, 10, 20, 50 or 100 concurrent requests). The concurrency level affects the number of requests received simultaneously by the framework. We carry out the experiments according tothe independent replication method with at least 5 iterations to achieve adequate statistical significance.

B. Impact of concurrent users

First, we measure the average response time and the ratio of successfully received responses under different levels of concurrent requests. Our aim is to isolate any performance issues due to the architecture of the framework itself. To this end, we write a simple function in Go that takes a string as input and sends the same string as the response. We choose this 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 auto scaling and run a fixed number of function replicas (1, 25 or 50) in each experiment. By doing so, we avoid possible increases in response times when scaling out functions, i.e., when creating new function pods/containers. We repeat each experiment 10 times to improve the accuracy of the results.

Figure 1 shows the median response time across all it- erations (i.e., 100,000 requests in total) for the different frameworks. The lowest median response time is achieved by Fission, with values around 2 ms in all scenarios. We observe that Kubeless and OpenFaaS maintain a response time below 80 ms across all scenarios. We also note that the response times do not show a significant change as the number of function replicas increases. In fact, functions deployed on Kubeless with 50 replicas for 100 concurrent requests obtain a response time slightly higher (by around 10 ms) than that with fewer function replicas. This indicates that it is possible to serve all requests for such a simple function with just one replica.

A closer examination of the results reveals that the response times for Fission have a significant number of outliers as the concurrency of

ISSN (PRINT): 2393-8374, (ONLINE): 2394-0697, VOLUME-4, ISSUE-1, 2017
DOI:10.21276/ijcesr.2017.4.1.2

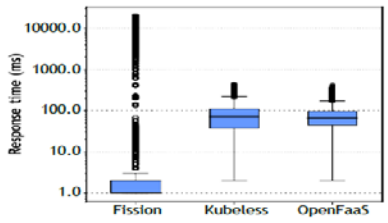

Fig. 2: Response time with one function replica and 100 concurrent requests.

Framework	Repl.	Number of concurrent users					
		1	5	10	20	50	100
Kubeless	1	100.00	100.00	100.00	100.00	100.00	100.00
	25	100.00	100.00	100.00	100.00	100.00	100.00
	50	100.00	100.00	100.00	100.00	100.00	100.00
Fission	1	100.00	99.90	99.84	99.78	99.54	99.32
	25	100.00	99.89	99.85	99.77	99.48	99.19
	50	100.00	99.88	99.81	99.79	99.61	99.31
OpenFaaS	1	99.95	99.99	99.91	99.58	98.73	98.27
	25	100.00	100.00	99.92	99.67	97.76	96.04
	50	100.00	100.00	99.93	99.61	97.48	96.52

TABLE II: Success ratio (in %) of all requests for different serverless frameworks.

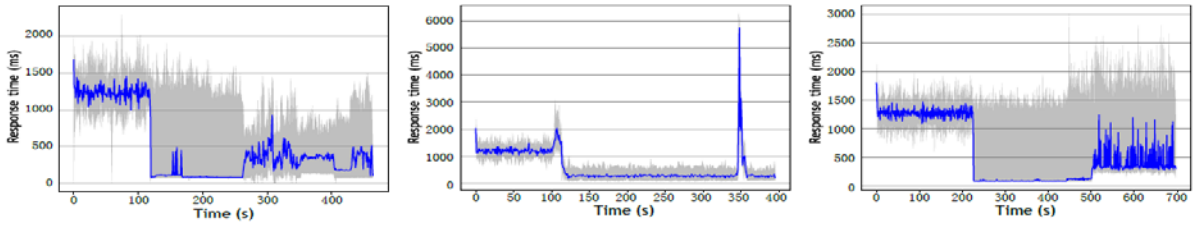

Fig. 3: Median response time as a function of time during auto scaling for (a) Fission, (b) Kubeless, and (c) OpenFaaS.

requests increases over 50. In this respect, Figure 2 shows the response times (on log scale) for 100 concurrent requests and one function replica for the three frameworks. OpenFaaS and Kubeless perform quite similarly and all responses are received within 400 ms. On the other hand, for Fission there are several outliers: 1,336 responses take more than 1 s and the longest response time goes up to 20 s. The large number of outliers for Fission pushes its average response time to 176 ms, whereas this behavior is not seen for OpenFaaS (74 ms) or Kubeless (79 ms). We also observe the same in other experiments with lower concurrency of requests and for higher number of function replicas. Thus, the performance of Fission deteriorates at high workloads regardless of the number of function replicas. We attribute this to the router component of Fission that forwards all incoming HTTP requests to the appropriate function. This component becomes a bottleneck as the workload increases. On the other hand, Kubeless relies on native Kubernetes components as far as possible: it utilizes the Kubernetes Ingress controller to route requests and balance the load. This component is at a more mature state, having been supported by Kubernetes since version 1.1 (available in 2015).

Next, we examine the ratio of successful responses for different levels of concurrency and number of function replicas (Table II). The table reports the success ratio over all ten iterations of the experiment. As the functions are invoked via HTTP, we consider any response without a 2xx response code as a failed request. We observe that Kubeless obtains the best performance with a 100% success ratio across all experiments, i.e., all HTTP responses were successfully received. Fission also manages to keep the success ratio at above 99% even at higher levels of concurrency. However, we observe that the success ratio of OpenFaaS drops to 98% or below when the number of concurrent requests is 50 or more. Furthermore, the success ratio is higher when only one function replica is present. This trend was seen consistently across multiple runs. We attribute this to the architecture of OpenFaaS wherein every function call has to go through multiple steps, resulting in many different points of failure. For instance, the HTTP requests and responses need to be processed by the gateway, faas-netes and the watchdog. Hence, the gateway and faas-netes can become bottlenecks (due to design or engineering issues) when the rate of incoming requests is high.

C. Impact of auto scaling

We now examine the impact of auto scaling on the re- sponse time and the ratio of successfully

ISSN (PRINT): 2393-8374, (ONLINE): 2394-0697, VOLUME-4, ISSUE-1, 2017
DOI:10.21276/ijcesr.2017.4.1.2

received responses. We choose to scale functions based on CPU utilization as Fission supports only this scaling metric. Accordingly, we consider a CPU-intensive function (in Go) that multiplies a 1,000 by 1,000 matrix on each invocation. This allows us to reach the CPU utilization threshold faster than with the previous function. We start each iteration of the experiment with a single function replica and set the threshold for CPU utilization at 50%. This implies that a CPU utilization exceed- ing 50% should trigger the creation of more function replicas. All frameworks use the Kubernetes Horizontal Pod Autoscaler

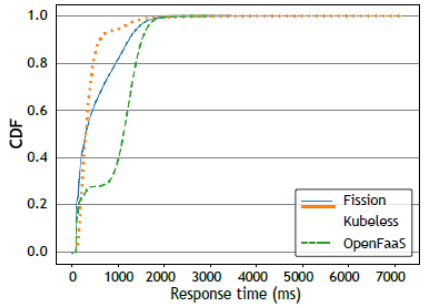

Fig. 4: CDF of response time (in milliseconds) with auto scaling enabled for each serverless framework.

components and its maturity. In fact, Kubeless has a version 1.0-alpha release whereas the other considered frameworks are at versions below 1.0. Clearly, all frameworks are under active development and are evolving rapidly. Nevertheless, our work is a first important step towards benchmarking the performance of serverless frameworks. Moreover, we note that some tuning is still required to achieve adequate performance, although serverless frameworks are expected to abstract away all scaling concerns from the developers [9]. With a simple “hello world” function,Kubeless and OpenFaaS maintain a low median and average response time below 80 ms. For more CPU-intensive functions, such as in the auto scalingscaling enabled for each serverless framework.(HPA)4 to perform scaling based on CPU utilization [20–22]. We use the ab tool to send 10,000 requests with 10 concurrent users and repeat each experiment 5 times.

All the frameworks leave the scaling decisions to the Ku- bernetes HPA. However, we notice that the ratio of successful responses and the distribution of response times varies between frameworks. Both Kubeless and OpenFaaS have a 100% success ratio across all experiments, whereas the success ratio for Fission is at 98.11%. Next, Figure 4 shows the distribution of response times over all the experiment runs. The median response time of OpenFaaS (1.1 s) is higher than the other two frameworks. Although Kubeless and Fission maintain a lower median response time (288 ms), the outliers reach a significantly higher value (up to 7 s). In fact, 50 responses (0.1%) took more than 3 seconds for Kubeless, whereas the occurrence of such outliers for other frameworks is below 5.

Next, we examine the variation of response time during a single iteration of the experiment. Accordingly, Figure 3 reports the values obtained by grouping all responses with a granularity of one second: their median response time as a solid line, as well as the corresponding minimum and maximum values as a gray band. In all cases, the median response initially lies between 1 s to 1.5 s. Both Kubeless and Fission are able to scale more replicas at approximately after 100 s of the experiment and thus, we see a reduced response time. However, Kubeless is able to maintain the low response time for a longer duration, whereas in the case of Fission the response time increases again after 260 s of the experiment run. OpenFaaS triggers a scaling request only after 200 s seconds of the experiment. We also note that the total duration of the experiment takes longer for OpenFaaS as the response time is quite high. This is because the ab tool waits for a response before sending more requests and the experiment only completes when all 10,000 requests have been sent.

D. Discussion

Our experimental results show that Kubeless has the most consistent performance across different scenarios. We attribute this to its simple architecture, the use of native Kubernetes4https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/experiments, the serverless framework itself may need to be scaled as well to avoid bottlenecks in individual components.

IV. RELATED WORK

ISSN (PRINT): 2393-8374, (ONLINE): 2394-0697, VOLUME-4, ISSUE-1, 2017
DOI:10.21276/ijcesr.2017.4.1.2

Serverless computing is receiving increasing attention in the academia [2, 7, 10, 43, 44]. Baldini et al. [2] summarize the general features of serverless platforms and describe open research problems in this area. Lynn et al. [10] present a feature analysis of seven enterprise serverless computing plat- forms, including AWS Lambda, Microsoft Azure Functions, Google Cloud Functions and OpenWhisk. Lee et al. [11] evaluate the performance of public serverless platforms by invoking CPU, memory and disk-intensive functions. They find that AWS Lambda outperforms other public cloud solu- tions. Furthermore, the authors highlight the cost-effectiveness of running functions on serverless platforms as compared to running them on traditional VMs. The authors also present a feature comparison of the public serverless platforms. Lloyd et al. [44] investigate the performance of functions deployed on AWS Lambda and Microsoft Azure Functions. They focus on the impact of infrastructure provisioning on public cloud platforms and identify variations in the functions' performance depending on the state (cold or warm) of the underlying VM or container. McGrath and Brenner [45] develop a prototype serverless platform implemented in .NET and using Windows containers for executing functions. The authors compare the performance of their prototype platform to AWS Lambda, Google Cloud Functions, Azure Functions and OpenWhisk. Shillaker [43] evaluates the response latency on OpenWhisk at different levels of throughput and concurrent functions. The author identifies research directions for improving start up time in serverless frameworks by replacing containers with a new isolation mechanism in the runtime itself. However, none of the works specifically address open source serverless platforms (Fission, Kubeless, OpenFaaS).

V. CONCLUSION

This article analyzed the status of open source serverless computing frameworks. First, we carried out a comprehensive feature comparison of the most popular frameworks, Fission, Kubeless, OpenFaaS and OpenWhisk. Based on that, we found that OpenFaaS has the most flexible architecture with support for multiple container orchestrators and easy extendability. Next, we evaluated the performance of Fission, Kubeless andOpenFaaS deployed on a Kubernetes cluster. Specifically, we characterized the response time and success ratio for functions deployed on these frameworks. We found that Kubeless has the most consistent performance across different scenarios. How- ever, all frameworks are under active development and changes are expected before the alpha release of each framework. As future work, we aim at analyzing the suitability of serverless computing for resource-constrained edge devices.

ACKNOWLEDGMENT

This work was partially supported by the Academy of Finland grant number 299222.

REFERENCES

[1] G. Adzic and R. Chatley, "Serverless computing: economic and architectural impact," in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017, pp. 884–889.

[2] I. Baldini et al., "Serverless computing: Current trends and open problems," in Research Advances in Cloud Computing. Springer, 2017, pp. 1–20.

[3] A. Kanso and A. Youssef, "Serverless: beyond the cloud," in Proceedings of the 2nd International Workshop on Serverless Computing. ACM, 2017, pp. 6–10.

[4] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," Future Generation Com- puter Systems, vol. 79, pp. 849–861, 2018.

[5] S. Nastic et al., "A serverless real-time data analytics platform for edge computing," IEEE Internet Computing, vol. 21, no. 4, pp. 64–71, 2017.

[6] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge comput- ing: extending serverless computing to the edge of the network," in Proceedings of the 10th ACM International Systems and Storage Conference. ACM, 2017, p. 28.

[7] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in

Proceedings of the 2017 Symposium on Cloud Computing. ACM, 2017, pp. 445–451.

[8] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell,

V. Muthusamy, R. Rabbah, and P. Suter, "Cloud-native, event- based programming for mobile applications," in Proceedings of the International Conference on Mobile Software Engineering and Systems. ACM, 2016, pp. 287–288.

[9] A. Eivy, "Be wary of the economics of "serverless" cloud computing," IEEE Cloud Computing, vol. 4, no. 2, pp. 6–12, 2017.

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

[11] H. Lee, K. Satyam, and G. C. Fox, "Evaluation of production serverless computing environments," in Proceedings of the 3rd International Workshop on Serverless Computing, 2018.

[12] "KubelessGitHub," https://github.com/kubeless/kubeless, (Accessed: 06/28/2018).

[13] "OpenFaaS: autoscaling," https://github.com/openfaas/faas, (Accessed: 06/30/2018).

[14] "Fission," https://github.com/fission/fission, (Accessed: 06/30/2018).

[15] "OpenWhisk," https://github.com/apache/incubator-openwhisk, (Accessed: 06/30/2018).

[16] "Kubeless runtime variants," https://kubeless.io/docs/runtimes/, (Accessed: 03/18/2018).

[17] "Openwhisk actions," https://github.com/apache/ incubator-openwhisk/blob/master/docs/actions.md, (Accessed: 06/27/2018).

[18] "Fission: Environments," https://docs.fission.io/0.8.0/concepts/ environments/, (Accessed: 06/30/2018).

[19] "OpenFaaS templates," https://docs.openfaas.com/cli/ templates/, (Accessed: 06/30/2018).

[20] "Kubelessautoscaling," https://github.com/kubeless/kubeless/ blob/master/docs/autoscaling.md, (Accessed: 06/27/2018).

[21] "Fission: Executors," https://github.com/fission/fission/blob/ master/Documentation/docs-site/content/concepts/executor.en. md, (Accessed: 06/26/2018).

[22] "OpenFaaS: autoscaling," https://docs.openfaas.com/ architecture/autoscaling/, (Accessed: 06/29/2018).

[23] "OpenWhisk deployment on Kubernetes," https:

//github.com/apache/incubator-openwhisk-deploy-kube, (Accessed: 06/27/2018).

[24] "faas-netes," https://github.com/openfaas/faas-netes, (Accessed: 06/29/2018).

[25] "faas-swarm," https://github.com/openfaas/faas-swarm, (Accessed: 06/29/2018).

[26] "faas-nomad," https://github.com/hashicorp/faas-nomad, (Accessed: 06/29/2018).

[27] "Kubeless architecture," http://kubeless.io/docs/architecture/, (Accessed: 03/18/2018).

[28] "Triggering IBM Cloud Functions on HTTP REST API calls," https://github.com/apache/incubator-openwhisk/blob/master/ docs/triggers rules.md, (Accessed: 06/27/2018).

[29] "Openwhisk: creating triggers and rules," https:

//github.com/apache/incubator-openwhisk/blob/master/docs/ triggers rules.md, (Accessed: 06/27/2018).

[30] "IBM Cloud Functions: your first action, trigger and rule," https:

//github.com/IBM/ibm-cloud-functions-action-trigger-rule, (Accessed: 06/27/2018).

[31] "Fission: Trigger," https://docs.fission.io/0.8.0/concepts/trigger/, (Accessed: 06/30/2018).

[32] "Openfaas watchdog," https://github.com/openfaas/faas/tree/ master/watchdog, (Accessed: 05/27/2018).

[33] "Queue worker for OpenFaaS - NATS Streaming," https://github.com/openfaas/nats-queue-worker, (Accessed: 06/29/2018).

[34] "Kubeless autoscaling," https://kubeless.io/docs/ pubsub-functions/, (Accessed: 06/30/2018).

[35] "OpenWhisk package for communication with Kafka or IBM Message Hub," https://github.com/apache/ incubator-openwhisk-package-kafka, (Accessed: 06/27/2018).

[36] "OpenFaaS: Kafka connector," https://github.com/ openfaas-incubator/kafka-connector, (Accessed: 06/29/2018).

[37] "Kubeless monitoring," https://kubeless.io/docs/monitoring/, (Accessed: 06/30/2018).

[38] "Fission: features," https://fission.io/features/, (Accessed: 06/30/2018).

[39] "OpenFaaS Workshop," https://github.com/openfaas/workshop, (Accessed: 06/29/2018).

[40] "OpenFaaS community," https://github.com/openfaas/faas/blob/ master/community.md, (Accessed: 06/29/2018).

[41] "Fission: Environments," https://docs.fission.io/0.8.0/concepts/ executor/, (Accessed: 06/30/2018).

[42] "Custom resources - kubernetes," https://kubernetes.io/docs/ concepts/extend-kubernetes/api-extension/custom-resources/, (Accessed: 06/27/2018).

[43] S. Shillaker, "A provider-friendly serverless framework for latency-critical applications," http://conferences.inf.ed.ac.uk/EuroDW2018/papers/eurodw18-Shillaker.pdf, (Accessed: 06/30/2018).

[44] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencingmicroservice performance," in Cloud Engineering (IC2E), 2018 IEEE International Conference on. IEEE, 2018, pp. 159–169.

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