A View On Open-Source Serverless Platforms

JETHA RAM (22EJCIT079)
Department of Information Technology
Jaipur Engineering College and Research Centre, Jaipur Engineering College and Research Centre
jetharam.it26@jecrc.ac.in,

Abstract:
Serverless computing is gaining popularity since it is cheaper as well as simpler to implement. There is a possibility for vendor lock-in, though. This is addressed by a number of open-source serverless solutions that allow developers to configure and manage operations on cloud servers they own. To build effective functions, one must be aware of the features and frameworks of the platform. Identifying the right serverless platform and distinguishing between requests and factors can be difficult for service developers. This study examines the event processing models and frameworks of four major open-source serverless platforms, as well as the differences in efficiency between auto-scaling and service exporting modes. The main objective of the following studies will be auto-scaling and metric gathering.

Keywords: serverless architectures, serverless; serverless computing, Function-as-a-Service, FaaS, Backend-as-a-Service, BaaS, AWS lambda, azure functions, google cloud functions, open whisk, stateful serverless, real-time serverless architecture, compare serverless offerings

1. Introduction

Serverless computing has revolutionized the field of cloud computing, offering large-scale and low-cost computing and storage services to end-users. This technology allows providers to commit only the required amount of resources to a particular application and utilize them for the time needed to execute an invoked function. Resource scaling is dynamically scaled to meet user demand, achieving nearly zero resource cost when there is no demand and scaling to as many instances as needed to meet traffic demand.

Serverless computing also allows developers to build, deploy, and run applications on demand without focusing on server management. The Cloud Native Computing Foundation (CNCF) states that when an event is triggered, a piece of infrastructure is allocated dynamically for function execution. Many cloud service providers offer serverless computing platforms on their public clouds, such as Amazon Web Services (AWS) Lambda, which enables developers to freely deploy and manage functions on self-hosted clouds.

To help developers choose suitable open-source platforms for efficient services, this paper systematically identifies and analyzes the salient characteristics of several popular open-source serverless platforms (Knative1, Kubeless2, Nuclio3, and OpenFaaS4) and compares their performance. Key contributions include understanding the platform frameworks and interaction between different components of these platforms, analyzing the salient features of each platform, evaluating the performance of different service exporting and auto-scaling modes, and analyzing the root cause of performance gaps among different serverless platforms.

The paper also provides insights for future work, such as auto-scaling and metric collection, to further enhance the capabilities of serverless computing in the cloud computing landscape.

Methodology

Reviewing Serverless Computing Methodology

- Search relevant databases like ACM Digital Library, IEEE Xplore, and Google Scholar for recent academic papers.
- Select relevant papers based on publication date, relevance to serverless computing, and source quality.
- Analyze selected papers to identify key themes like benefits, challenges, and future predictions.
- Categorize and synthesize findings for a comprehensive understanding.
- Structure the information into a review paper, including background, current state, future trends, conclusions, comparisons of cloud providers' offerings, and discussions of real-world applications.

2. Experiments and Results

Serverless computing platforms, such as AWS Lambda, Google Cloud Functions, Azure Functions, and Alibaba Cloud Function Compute, are provided by cloud service providers (CSPs) on their public clouds. Since developers must rely the design and implementation of their serverless functions on the services provided by these CSPs, there may be vendor lock-in and challenges when transferring existing functions to other public clouds or self-hosted clusters. Open-source serverless solutions give developers greater flexibility by enabling them to deploy and run apps on self-hosted clouds. These platforms do, however, have several drawbacks,

Feature	Nuclio	OpenFaaS	Knative	Kubeless
Queue inside Function Pod	✔	✔	✔	✖
Support for Multiple Workers in Function Pod	✔	✖	✔	✖
Function Startup Policy	Warm Start	Cold/Warm Start	Cold/Warm Start	Cold Start
Service Export Method	Ingress Gateway/NodePort	API Gateway/Ingress Gateway	Ingress Gateway	Ingress Gateway
Runtime Metric Collection	Metric Server	Metric Server/API Gateway	Metric Server/Queue-proxy	Metric Server
Auto-scaling Mode	CPU/Memory	CPU/Memory/RPS	CPU/Memory/Concurrency/RPS	CPU/Memory
Scale-to-zero	✖	✖	✔	✖

TABLE I: Comparision of popular open-source serverless platforms

such as the requirement of a comprehensive understanding of the features, autonomous administration and maintenance, and inconsistent behavior across different environments. Therefore, it is essential to review the features of widely recognized open-source serverless platforms and contrast their performance in order to help developers choose the right platforms.Our results demonstrate that our system achieves competitive performance in terms of summarization quality and computational efficiency.

3. Discussion

Based on recent popularity, community vibrancy and feature richness, we specifically select four open-source serverless frameworks, *i.e.,* Knative, Kubeless, Nuclio and OpenFaaS, to analyze their characteristics.

A. *Dependency on Kubernetes*

Kubernetes is a portable and extensible open-source system that facilitates declarative configuration, automating deployment and management for containerized workloads. Most of open-source serverless platforms rely on Kubernetes for orchestration and management of function pods, which are the atomic deployable units in Kubernetes. Fig. 1 shows the pivotal Kubernetes services that serverless platforms depend on. These Kubernetes services are used for: (1) configuration management, (2) service discovery, (3) auto-scaling, (4) pod scheduling, (5) traffic load balancing, (6) network routing and (7) service roll-out and roll-back.

Thanks to the horizontal pod auto-scaler (HPA) feature from Kubernetes, the Kubernetes-based serverless platforms support resource-based auto-scaling. The framework of HPA is shown in Fig. 2. The Kubelet on each node collects the resource metrics of each pod. HPA gets these metrics from the API server. The auto-scaling threshold could be a raw value or a percentage of the pod requested amount for that resource. When the CPU or memory usage of a given function pod exceeds the threshold, HPA automatically triggers the Development controller to scale the pod number.

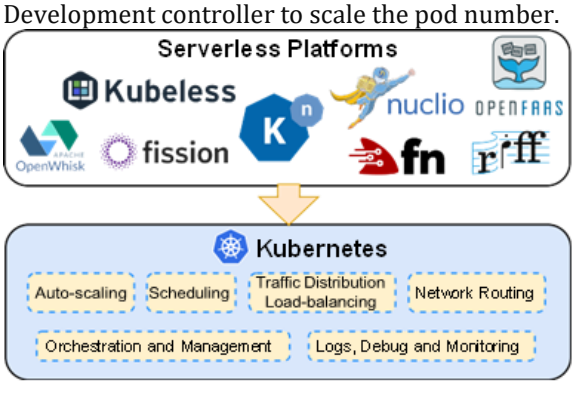


Fig. 1: Serverless platforms and underlying cubernetes ser- vices.

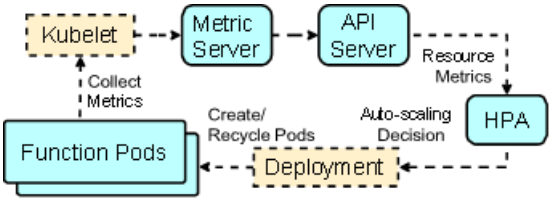


Fig. 2: Horizontal pod auto-scaling framework.

B. Salient Features of Serverless Platforms

Table I summarizes salient features of four widely-known open-source serverless platforms.

1) Nuclio: The main components of Nuclio are shown in Fig. 3. In each function pod, there is one event listener and multiple worker processes. The event listener receives new events and redirect them to worker processes. Multiple worker processes could work in parallel and improve the performance significantly on a multi-core worker node. The worker process number is set to be static and specified by the configuration file. The open-source version does not have a built-in workload-based auto-scaling feature, but the resource-based auto-scaling is supported by Kubernetes HPA.

Nuclio supports two ways to trigger functions: (1) invoking the function by name through ingress controller, which can distribute the traffic to different back-end pods according to the pre-set load balancing rule (e.g., round-robin, random and least connection first) and (2) sending requests directly to function pods by NodePort, which is a unique allocated cluster-wide port for the function. In the NodePort method, incoming requests are load balanced at random by Netfilter.

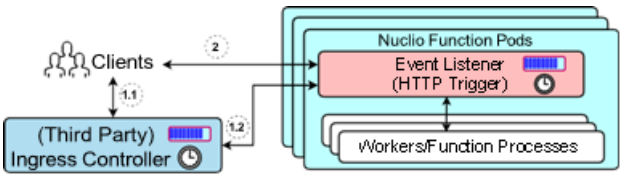


Fig. 3: Nuclio framework.

2) OpenFaaS: The key components of OpenFaaS are shown in Fig. 4. The API gateway provides access to the

functions and collects traffic metrics. Faas-netes is the controller for managing OpenFaaS function pods. Prometheus5 and AlertManager6 are used for auto-scaling.

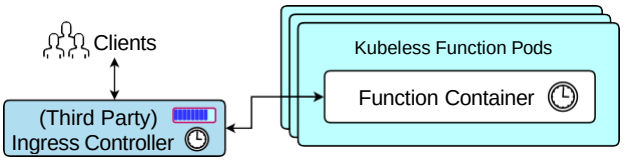


Fig. 4: OpenFaaS framework.

Each function pod contains a single container running two type of processes, namely of-watchdog and function process. Of-watchdog is a tiny server that works as the entry-point for incoming requests and forwards them to the function process. Of-watchdog can operate in three modes, i.e., HTTP, streaming and serializing. In HTTP mode, the function process is forked only once at the beginning and kept warm for the entire life cycle of the function pod. In both the streaming and serializing mode, a new function process is forked for every request, resulting in significant cold-start latency and adverse impact on performance. Our evaluation results show that the throughput of the streaming or serializing mode is about 10× lower than that of HTTP mode.

OpenFaaS has a built-in requests-per-second (RPS) based auto-scaling feature. Prometheus scrapes the traffic metrics from API gateway. AlertManager reads the RPS metric from Prometheus and fires an alert to the API gateway according to the auto-scaling rule defined in the configuration file. Then the API gateway handles the alert and invokes the Faas-netes to scale up or scale down function replicas. Note that the open- source version does not support scale-to-zero feature, which is only available in the commercial version, i.e., OpenFaaS Pro.

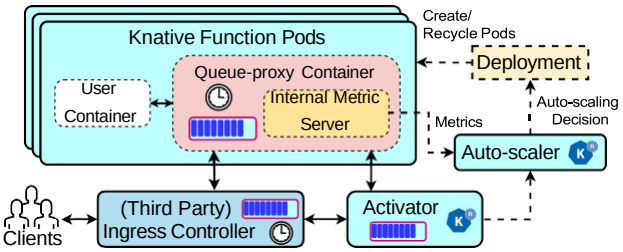
3) Knative: Fig. 5 shows the Knative framework. Each function pod consists of two containers, namely queue-proxy container and function container. The queue-proxy is a sidecar container to queue incoming requests and forward them to the function container. The queue-proxy provides a buffer to handle traffic burst in spite of incurring queuing latency. In addition, the queue-proxy collects metrics and expose them via a simple HTTP server, i.e., internal metric server. Multiple workers reside in the user container to process requests in parallel. The communication overhead between queue-proxy container and function container is higher than the process model of Nuclio and OpenFaaS, and thus results in lower performance.

The Knative built-in auto-scaling, i.e., Knative pod autoscaler (KPA), supports both RPS mode and concurrency mode. The auto-scaler scrapes metrics from function pods and computes the replica number based on the auto-scaling algorithm. The deployment controller gets the auto-scaling de- cision and adjusts the pod number. Knative supports scale-to- zero functionality which recycles all pods of inactive functions. When a new request arrives for an idle function, the ingress controller redirects the request to the activator to buffer it. Then the activator triggers the autoscaler which could scale up the idle function from zero. Once the function is running again, the activator sends the buffered request to the pod. Although scale-to-zero reduces resource usage, it leads to extra cold start latency.

4) Kubeless: Kubeless is another open-source platform built on top of Kubernetes. Fig. 6 describes the key compo- nents and the working model of Kubeless. There are several options for ingress controllers. We experiment with Nginx ingress controller7 and Traefik ingress controller8, and opt for Traefik due to better performance. Kubeless leverages Kubernetes HPA for auto-scaling and does not support scale- to-zero.

Fig. 6: Kubeless framework.

4. Applications of Text



Summarization System

We first compare the overall performance of different open- source serverless platforms. Based on the performance results of multiple service exporting modes and auto-scaling modes, we analyze the root cause of performance gap among different serverless platforms.

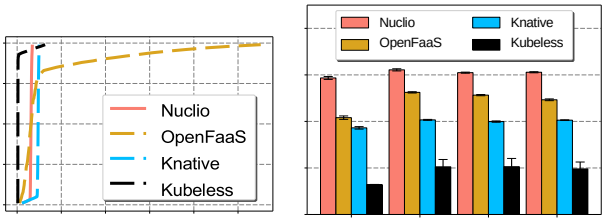
A. Experimental Setup and Workload Description

We evaluate the serverless platforms on the CloudLab testbed [6] consisting of one master and two worker nodes, each of them equipped with Intel CPU E5-2640v4@2.4GHz (10 physical cores), running Ubuntu 16.04.1 LTS (kernel 4.4.0-154-generic). We build all four serverless platforms on Kubernetes (v1.20.0), using the latest version available at the time of writing9. Several serverless functions of Python 3.6 runtime are implemented. We use wrk10 to generate HTTP workloads for invoking serverless functions.

B. Performance

1) HTTP Workload: To evaluate the baseline performance of different serverless platforms, we implement a HTTP work- load function that could fetch a four-byte static webpage from a local HTTP server on the master node. For a fair comparison, we limit to a single instance of the function pod, disable auto-scaling and configure the same queue size and timeout parameters (50K requests, and 10s timeout) at the ingress gateway and function pod components across all the platforms. For Nuclio and Knative, we further restrict it to a single worker in one pod. Every experiment lasts for two minutes and we measure for one minute after one-minute warm-up. The experiment is repeated for 20 times. Fig. 8 shows the throughput for varying number of concurrent connections and the latency profile for concurrency level of 100. Nuclio has the least 99%ile latency within 500ms, as it allows queuing only within the function pod, while OpenFaaS and Knative can queue requests at ingress/gateway components. OpenFaaS shows heavy tail due to queuing at both the gateway and of- watchdog components. Kubeless drops the connections at the ingress, resulting

in additional retries from the client and hence lower throughput. The latency with Kubeless is lower because there is no queue inside the Kubeless function pod.



(a)latency(concurrency=100). (b)Throughput.

Fig. 7: Performance of HTTP workload function. Error bars indicate standard deviation over 20 runs.

2) Latency Breakdown of Single Request: We analyze the delay overheads incurred in processing serverless functions for different platforms. We breakdown the processing delays within the function pod. For this experiment, we use curl to send one request for hello-world function[11] and use tcpdump to capture the packets on the worker node of the function pod. We record four timestamps, *i.e.,* (1) when the request reaches the function pod; (2) start of the function runtime; (3) end of the function runtime; (4) when the response is sent out of function pod. The experiment is repeated for 20 times and the average time intervals between these timestamps are shown in Fig. 9. In all frameworks, the actual run-time of the function (0.001ms) is the same. However, the function initiation time (time taken for request to be forwarded to the function instance) and function response delay (time taken for the response of the function to be sent out of the pod) vary. This depends on how the data is packaged and shared with the function instance. Due to forking-per-request, Kubeless incurs very high delay in forwarding the packet to the function instance.

Process	1→2	2→3	3→4
Nuclio	0.63	0.001	0.54
OpenFaaS	1.32	0.001	0.93
Knative	1.30	0.001	0.62
Kubeless	4.96	0.001	2.63

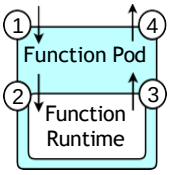


Fig. 9: Latency breakdown of function execution (ms).

5.

1) Promising Platform: In spite of the moderate perfor- mance compared with other open-source platforms, Knative has many useful features, such as scale-to-zero and multiple auto-scaling modes, and active community that can provide lots of help for users and developers. Thus Knative is a suitable platform for further development and innovation in serverless computing.

2) Auto-scaling: For current auto-scaling mechanisms, such as workload/resource-based auto-scaling, the auto-scaling met ric and threshold are set by tenants. Because the tenants may not really know the runtime characteristics of their functions (*i.e.,* resource usage and execution time), they easily mis-configure the auto-scaling settings. In addition, it is not always easy to predict the correct indicators that could show whether the current function pods are under-resourced or under-utilized. Thus a more smart auto-scaling algorithm is needed to be designed to both properly meet the workload demand and save the resources in different scenarios.

2) Function Startup Policy: There are two options for function startup policy, *i.e.,* cold start and warm start. For the functions with low invocation rates, cold start policy could help reduce resource usage in the case of no incoming requests, but it leads to extra cold start latency. Therefore, the cold start is not appropriate for time-sensitive functions [8]. We should carefully choose the function startup policy in accordance with the scenarios and user demands.

3) Metric Collection: The on-demand provisioning feature of serverless computing depends on several mechanisms, such as auto-scaling, scheduling and load balancing. All these mechanisms leverage metrics to make the decision. Many plat- forms, such as Knative and OpenFaaS, use scraping method to fetch metrics from pods. In our experiments, we find that the scraping method may leads to large traffic overhead when there are a large number of pods. Using sampling to only scrape a section of pods can partly decrease the overhead. However, sampling may miss out the abnormal pods and reduce the accuracy of metrics. Hence, a more efficient metric collection mechanism is worth studying further.

4) Service Export and Network Routing: There are many ways to export services and route incoming requests to back- end function pods, such as cluster IP, NodePort, function name/URL. All of them have both strengths and weaknesses, and should be chosen with caution.

5) Function Chain: It is useful to chain multiple functions for stateful workflows and complex services. How to make function chain more efficient and powerful needs to be ex- plored in future work.

6. Conclusion

We elaborate the working models of different popular opensource serverless platforms and identify their key characteristics. In addition, we analyze the root causes of performance gap of different service exporting and auto-scaling modes on those platforms. Further, several insights are proposed for future work, such as auto-scaling, service export and metric collection.

7. References

[1] J. Li, S. G. Kulkarni, K. Ramakrishnan, and D. Li, "Understanding Open
Source Serverless Platforms: Design Considerations and Performance,"
in Proceedings of the 5th International Workshop on Serverless Computing (WoSC), 2019, pp. 37–42.

[2] B. Burns and et al, “Borg, Omega, and Kubernetes,” Commun. ACM, vol. 59, no. 5, pp. 50–57, 2016.

[3] J. Li, D. Li, Y. Huang, Y. Cheng, and R. Ling, “Quick NAT: High Per- formance NAT System on Commodity Platforms,” in IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN). IEEE, 2017, pp. 1–2.

[4] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide,

L. Stoller, M. Hibler, D. Johnson, K. Webb et al., “The Design and Operation of CloudLab,” in USENIX Annual Technical Conference (ATC), 2019, pp. 1–14.

[5] J. Li, D. Li, Y. Yu, Y. Huang, J. Zhu, and J. Geng, “Towards Full

Virtualization of SDN Infrastructure,” Computer Networks, vol. 143, pp.

1–14, 2018.

[6] J. Li, D. Li, W. Wu, K. Ramakrishnan, J. Geng, F. Gui, F. Wang, and

K. Zheng, “Sphinx: A Transport Protocol for High-Speed and Lossy Mobile Networks,” in IEEE 38th International Performance Computing and Communications Conference (IPCCC). IEEE, 2019, pp. 1–8.

[7] Y. Huang, J. Geng, D. Lin, B. Wang, J. Li, R. Ling, and D. Li, “LOS: A High Performance and Compatible User-level Network Operating Sys- tem,” in Proceedings of the First Asia-Pacific Workshop on Networking (APNet), 2017, pp. 50–56.

[8] G. McGrath and P. R. Brenner, “Serverless Computing: Design, Imple- mentation, and Performance,” in IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2017, pp. 405–410.

[9] L. Wang and et al, “Peeking Behind the Curtains of Serverless Plat- forms,” in USENIX Annual Technical Conference (ATC), 2018, pp. 133– 146.

[10] J. Wen, Y. Liu, Z. Chen, Y. Ma, H. Wang, and X. Liu, “Understanding Characteristics of Commodity Serverless Computing Platforms,” arXiv preprint arXiv:2012.00992, 2020.

[11] W. Lloyd and et al, “Serverless Computing: An Investigation of Factors Influencing Microservice Performance,” in IEEE International Confer- ence on Cloud Engineering (IC2E). IEEE, 2018, pp. 159–169.

[12] S. Allen and et al., “CNCF Serverless Whitepaper,” https://github.com/

cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf

serverless whitepaper v1.0.pdf, 2018, [ONLINE].