Understanding Open Source Serverless Platforms: Design Considerations and Performance

Junfeng Li[1,2], Sameer G. Kulkarni[2], K. K. Ramakrishnan[2], and Dan Li[1]
[1]Tsinghua University [2]University of California, Riverside

Abstract
Serverless computing is increasingly popular because of the promise of lower cost and the convenience it provides to users who do not need to focus on server management. This has resulted in the availability of a number of proprietary and open-source serverless solutions. We seek to understand how the performance of serverless computing depends on a number of design issues using several popular open-source serverless platforms. We identify the idiosyncrasies affecting performance (throughput and latency) for different open-source serverless platforms. Further, we observe that just having either resource-based (CPU and memory) or workload-based (request per second (RPS) or concurrent requests) auto-scaling is inadequate to address the needs of the serverless platforms.

CCS Concepts • **Networks → Network measurement**; **Cloud computing**; Network design principles.

Keywords serverless, function-as-a-service, performance

ACM Reference Format:
Junfeng Li[1,2], Sameer G. Kulkarni[2], K. K. Ramakrishnan[2], and Dan Li[1]. 2019. Understanding Open Source Serverless Platforms: Design Considerations and Performance. In *Fifth International Workshop on Serverless Computing (WOSC '19), December 9–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3366623.3368139

1 Introduction
Serverless computing has ushered in a new era in cloud computing. Cloud computing seeks to provide compute and storage services at large scale and low cost to end-users through economies of scale and effective multiplexing. Serverless computing takes this multiplexing and scalability to the next level by allowing providers to commit just the required amount resources to a particular application (as many instances as necessary, but only when needed) and utilize the resources for just the time needed to execute an invoked function. Resources are scaled dynamically to meet the demand from user requests. Unlike the 'traditional' cloud deployment model, where the number of necessary compute instances are deployed well in advance, serverless computing allows the cost to be near zero when there is no demand, and scales to as many instances as needed to meet the traffic demand. Thus, serverless is meant to be both scalable and more cost effective.

In addition to scaling and multiplexing, serverless computing allows developers to build, deploy and run the application on demand without focusing on server management, according to the Cloud Native Computing Foundation (CNCF) [2]. When an event is triggered, a piece of infrastructure is allocated dynamically to execute the code. The underlying details of resource management: resource allocation, communication of user data, and the execution of functions is abstracted from the user. Serverless computing manages cloud resources typically by deploying applications in dynamically instantiated containers. For instance, Amazon provides AWS-Lambda [3], an event-driven, serverless computing platform that enables to implement and deploy application code in any of the supported languages and execute on-demand as docker-containers. The serverless infrastructure manages the queuing of requests and can automatically scale containers to meet fluctuating demands.

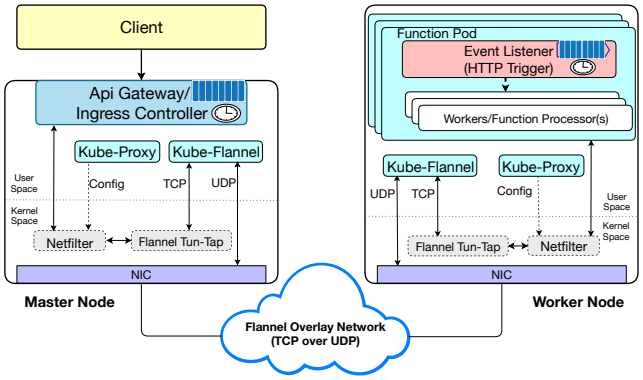


Figure 1. Kubernetes: Network routing to export the services.

Our focus is not only on the evaluation and comparison of performance, but seek to identify the key differences in the workings of different Kubernetes-based open-source serverless platforms. We systematically identify the strengths and deficiencies of Knative[1], Kubeless[2], Nuclio[3] and OpenFaaS[4]. Our key contributions include:

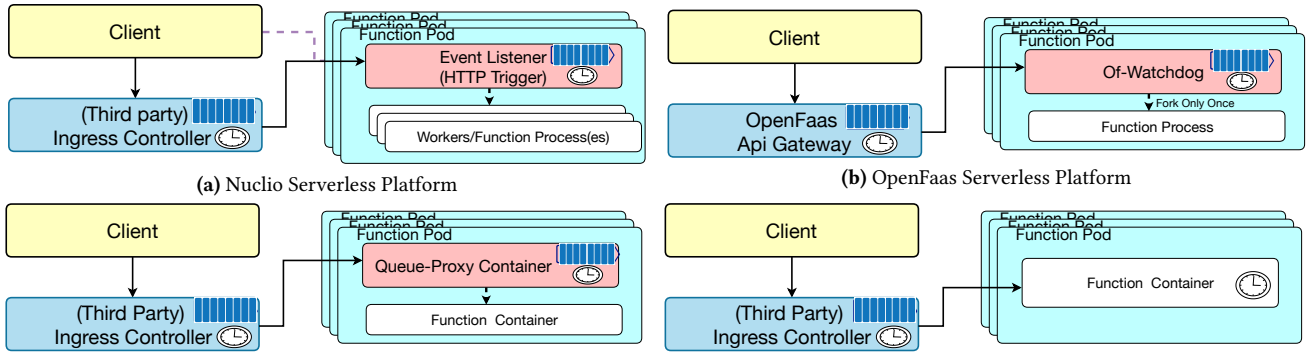
- We provide an understanding of the role and interaction of the different components of each of these platforms.
- We describe the impact of key configuration parameters of different components (platform, gateway, controller and function).
- We evaluate the mode and operation of auto-scaling supported by these different platforms for different kinds of workloads.

2 Background & Comparison
Several cloud service provides (CSPs) offer serverless computing platforms on their public clouds *e.g.,* AWS Lambda functions, Google Cloud Platform, Microsoft Azure, and IBM Bluemix *etc.* These cloud platforms also offer other supporting services such as an event notification service, storage service, database services *etc.* that are necessary for operating an overall serverless ecosystem. These CSPs govern many of the function-related characteristics such as: how long functions can run, how long can they be kept idle, the number of concurrent active instances, load balancing among the active instances, retry in the case of failed requests *etc.* These are almost entirely dependent on the cloud providers' terms and conditions. To understand the impact of these choices, it is useful to study the

[1] https://github.com/knative [2] https://kubeless.io [3] https://nuclio.io
[4] https://www.openfaas.com

WOSC '19, December 9–13, 2019, Davis, CA, USA

ACM ISBN 978-1-4503-7038-7/19/12…$15.00
https://doi.org/10.1145/3366623.3368139



(a) Nuclio Serverless Platform **(b)** OpenFaas Serverless Platform

(c) Knative Serverless Platform **(d)** Kubeless Serverless Platform

Figure 2. Working model for different kubernetes-based serverless platforms (Nuclio, OpenFaas, Knative and Kubeless).

functioning of open source serverless platforms such as Knative, Kubeless, Nuclio, OpenFaaS, OpenWhisk[5], *etc.*

2.1 Open-source Serverless platforms

Several open source serverless platforms allow us to freely leverage and mix-and-match different open source services, and to deploy and manage the functions on self-hosted clouds. However, the challenges are the i) readiness (requires learning and setup expertise) of the necessary infrastructure and integration of different services, ii) challenges with management and maintenance of the necessary service infrastructure. iii) lack of technical support. Hence, in this work we specifically select four of the Kubernetes [4] based open source serverless frameworks based on the recent popularity,[6] community support and feature richness of these platforms.

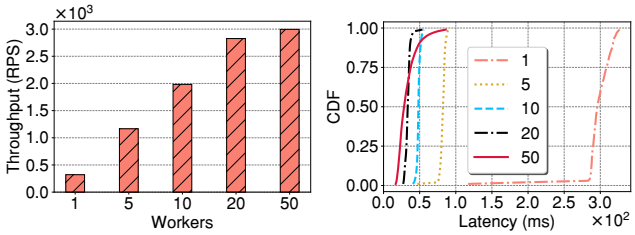
2.2 Dependency on Kubernetes

Kubernetes is a portable and extensible platform that facilitates both declarative configuration and automation of deployment and management of containerized workloads. The serverless frameworks rely on Kubernetes APIs for orchestration and management of the serverless functions. Serverless platforms typically extend and provide the Custom Resource Definition (CRD) features necessary to create and deploy the container pods (group of containers). They depend primarily on Kubernetes for i) Configuration management of containers and pods; ii) Pod scheduling and service discovery; iii) Update roll-outs for functions; and iv) Replication management.

2.3 Salient Characteristics of Serverless Platforms

Fig. 2 shows the framework and key components of the 4 different serverless platforms considered in this work.

Nuclio: Fig. 2a shows the key components of Nuclio. The distinct feature of Nuclio is the 'Processor' architecture which provides work parallelism through multiple worker processes that can run in each container. First, the Nuclio service model supports invocation of the 'function' pod directly from an external client, without the need for any ingress controller or API gateway. Second, the function pod consists of two kinds of processes namely the i) event-listener and ii) one or more worker (user deployed function) processes. Note, the event-listener can be configured with a timeout parameter to control how long events can be queued. Third, the number of worker processes can be setup as a static configuration parameter. This enables the function pod to run a desired amount of function instances as different processes, and allows parallel execution on a multi-core node.



(a) Throughput in requests/second. **(b)** Latency in ms.

Figure 3. Throughput and latency for different number of workers within one Nuclio function pod (100 concurrent requests).

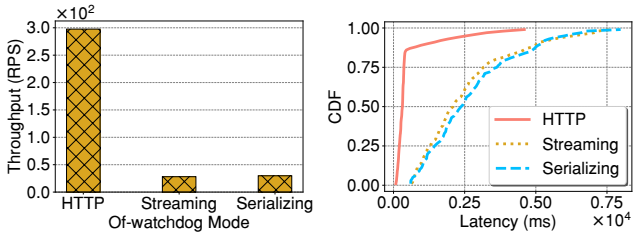
To quantify the benefit of having multiple workers, we experimented with simple 'http-workload' where we implement a simple python function that communicates with a local HTTP server (located on Kubernetes master node), to fetch and respond with a 4 byte payload for each of the requests. Fig. 3a shows the impact on throughput and latency for multiple workers. We observe a 4× throughput increase with 4 workers and almost 10× improvement with 50 workers. Note, scaling the number of workers also improves the latency as shown in Fig. 3b.

OpenFaaS: The key components of OpenFaaS are shown in Figure 2b. The API gateway provides access to the functions from outside the Kubernetes cluster (external routing), collects metrics and provides scaling by interacting with the Kubernetes orchestration engine. The API gateway can be scaled to multiple instances. Also, it can be replaced by a third-party Ingress controller.

Each function pod consists of a single container running two kinds of processes namely the i) 'of-watchdog' and ii) user deployed function process. The 'of-watchdog' is a tiny Golang HTTP server that serves as the entry-point for HTTP requests to be forwarded to the function process. Based on use case requirements, the 'of-watchdog' can be operated in 3 modes, i.e., 'HTTP', 'streaming' and 'serializing'. In 'HTTP' mode, the function is forked only once to one instance (worker) 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 instance (worker) is forked for every request, resulting in significant cold-start latency and impact on the throughput. Fig. 4 shows the throughput and latency when running the watchdog in different supported modes. The 'streaming' mode results in very low performance and is typically only desirable for memory-heavy workloads, while the 'serializing'

[5] https://openwhisk.apache.org [6] Until the release of Knative and Nuclio, the Kubeless and OpenFaaS were shown to be the top two leading serverless platforms in terms of current and planned usage [10]

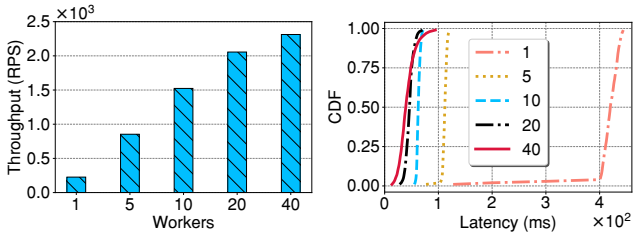
mode is equally poor due to fork per request. For our subsequent evaluation, we choose the 'HTTP' mode.



(a) Throughput in requests/second. **(b)** Latency in ms.

Figure 4. Throughput and latency for different modes of OpenFaaS of-watchdog (100 concurrent requests).

Knative: Fig. 2c, shows the key components of Knative. We see that each function pod consists of two containers namely the 'queue-proxy' and the 'function'. The 'queue-proxy' is responsible for queuing incoming requests and forwarding them to the 'function' container for execution. It also handles the timeout of queued requests. This queue enables the worker to quickly fetch requests from the ingress controller and process them, thus achieving better throughput, although incurring queuing latency. Interestingly, we can observe that since Knative implements the 'queue-proxy' and 'function' as two different containers in a pod, the communication overhead is higher than the process model of the Nuclio and OpenFaaS, resulting in relatively lower performance.



(a) Throughput in requests/second. **(b)** Latency in ms.

Figure 5. Throughput and latency for different number of workers within one Knative function pod (100 concurrent requests).

For the python runtime, we observed that Knative levarages gunicorn[7] - a Python web server gateway interface (WSGI) server, which supports the pre-fork worker model to create multiple worker processes in a function pod. However, unlike Nuclio, the number of concurrent workers is not exported as a configuration parameter for deployment. Figure 5 shows the impact on throughput and latency for multiple workers. The characteristics are similar to the Nuclio workers, discussed earlier.

Another distinct feature of Knative is the 'panic mode' scaling mechanism of the autoscaler component. Panic mode enables the autoscaler to be more responsive to sudden traffic spikes (two times the desired average traffic or a configured threshold value) by quickly scaling the functions instances (up to 10× the current pod count or the maximum configured limit).

Kubeless: Kubeless is another open source platform built on top of Kubernetes. Figure 2d describes the key components and the working model of serverless functions in Kubeless. We also experimented with NGINX ingress controller,[8] and opted for Traefik due to better performance.

[7] http://gunicorn.org [8] https://kubernetes.github.io/ingress-nginx

2.3.1 Exporting Services and Network routing

Serverless frameworks leverage Kubernetes network model to export services (cluster of function pods) and to route requests to specific functions. An API Gateway and/or the Ingress Controller components of the serverless platform can be either exported with a public IP address or can also use the Kubernetes networking model to export the services. Fig. 1 describes 'Flannel' - a simple Kubernetes overlay networking framework to export serverless functions and route the traffic to function pods. The Kube-Proxy component of Kubernetes is responsible for setting up the routing and load-balancing rules (*e.g.,* setup the netfilter rules to intercept network packets and change their destination/routing) of the traffic intended for Kubernetes pods, while the Kube-Flannel pod is responsible for intercepting the packets destined for Kubernetes pods (listen to traffic for the virtual Kubernetes pod IP range) and performing UDP encapsulation/decapsulation for the traffic exiting/entering the physical network interface. In Fig. 1, once the API Gateway/Ingress Controller receives client packets, and determines the service (function) to be executed, it leverages the Kube-Proxy and Kube-Flannel to load-balance and route the traffic to a specific function pod of a worker node. With Kube-Flannel, the traffic leaving the physical network interface is encapsulated and carried over an unreliable UDP transport.

Impact of Ingress Controller and API Gateway components: Typically, the API Gateway components enable the URL based routing to different services in a Kubernetes cluster. The function pods are dynamic entities that can be created and destroyed any time because of zero-scaling, auto-scaling, failures *etc.* Hence, Kubernetes provides service (a virtual cluster with fixed IP, *a.k.a.* 'Cluster IP') as an abstraction to access the pods of a similar kind. The API Gateway/Ingress controllers can route the incoming requests in two possible ways: i) route the incoming traffic to the service and let Kubernetes control load-balancing of the traffic across active pods (*e.g.,* with the OpenFaaS API Gateway); ii) load-balance and route the traffic directly to any of the active pod instances (*e.g.,* with the Knative-Istio ingress controller).

In the former case (API Gateway), we observed that, in order to avoid the overhead of connection setup time, the API Gateway (OpenFaaS API Gateway) sets up multiple connections with the service 'Cluster IP'[9] at the beginning (the first access to the function) and it uses these connections to forward subsequent requests. No new connections are setup afterwards, unless the existing connections get terminated. Note that if the connections are not setup after auto-scaling, the traffic cannot get distributed to the newly created pods, thus significantly impacting the performance with auto-scaling (refer §3.3). However, in the second case (case ii), the ingress controller needs to keep track of the health and status of all the active pods and setup the connections explicitly with each of the active pods to load-balance the traffic.

3 Evaluation

The main focus of our evaluation is to distinguish and illustrate the impact of the serverless platform specific design choices and their dependency on the Kubernetes orchestration and management services. A second important focus is to understand the auto-scaling capabilities, and the need to go beyond the resource utilization based scaling services provided by Kubernetes.

[9] Service being a logical entity, the actual TCP connections are setup with different active pods based on the Kubernetes routing/load-balancing rules (*e.g.,* netfilter rules).

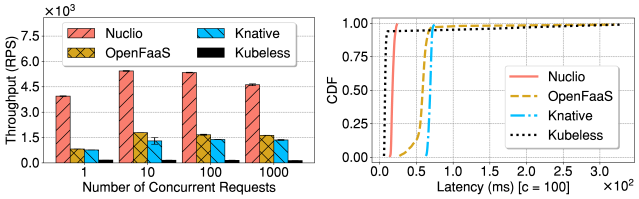
3.1 Experimental setup and Workload description

We evaluate the serverless platforms on the Cloudlab testbed [9] 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 built all four serverless platforms on Kubernetes (v1.15.3), using the latest version available at the time of writing.[10] We choose Python 3.6 and implement different serverless functions viz. i) simple Hello-world function as the baseline, and ii) HTTP server function that fetches and serves pages of different sizes from the local HTTP server. We use `wrk` [1] to generate the HTTP workloads and invoke the serverless functions.

3.2 Performance - Throughput and Latency

3.2.1 Baseline Performance

To evaluate the baseline performance *i.e.,* throughput (average requests processed per seconds) and response latency of different serverless platforms, we use a simple 'Hello-world' - a no operation function, that returns 4 bytes of static text in the response. For a fair comparison, we limit to a single instance of the function pod, disable auto-scaling and configured 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, we further restricted it to a single worker process.



(a) Avg. throughput. **(b)** Latency for concurrency of 100.

Figure 6. Throughput and latency of 'hello-world' function.

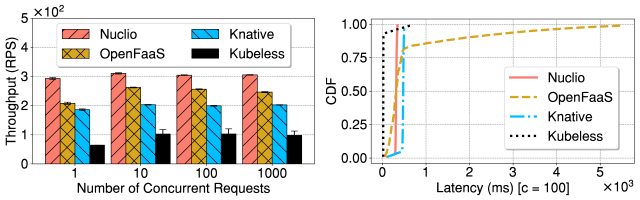
Fig. 6a shows the baseline throughput achieved by different platforms for different concurrent executions of requests. Nuclio outperforms the other platforms due to the low overhead of a direct function call. Routing through the API gateway/Ingress controller components incurs not just the overhead for HTTP connection termination, but also for the context-switch/transfer of the packets across the kernel and user-space of the worker node twice to get it routed to the function pod as shown in the Fig. 1. To quantify the overhead, we also experimented with Nuclio using an ingress controller mode and observed the overhead. It resulted in almost half the throughput (~1700 RPS as opposed to ~3000 RPS for direct call) and nearly 2× latency overhead (increases from 356μs to 611μs). At the other extreme, Kubeless forks the function for every request, resulting in severely degraded throughput and latency.

From Fig. 6b, we can observe that median latency is lowest for Kubeless, and is marginally higher (20~50 ms) for the queue based frameworks. However, tail latency (above 95%ile) degrades severely for Kubeless and OpenFaaS, while Nuclio and Knative do not see this increased heavy tail-latency. The results indicate that having process based communication within a container (*e.g.,* Nuclio) along with a local worker queue achieves better throughput by having lower overhead for processing requests.

[10] Nuclio (v1.1.16). OpenFaaS consists of: Gateway (v0.17.0), Faas-netes (v0.8.6), Prometheus (v2.11.0), Alert manager (v0.18.0), Queue worker (v0.8.0) and Faas-cli (v0.9.2), and the HTTP mode of-watchdog. We use Knative (v0.8) with Istio (v1.1.7) ingress controller, and Kubeless (v1.0.4) with Traefik ingress controller (v1.7).

3.2.2 HTTP Workload

Next, we change to having http-workload.Again, we keep the serverless platform settings the same as described in the baseline experiment §3.2.1. Fig. 7 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 watchdog components, each having distinct timeout parameters. Kubeless drops the connections at the ingress, resulting in additional retries from the client - hence it's lower throughput (the lower latency with Kubeless is because it is measured only for those requests that succeed at a concurrency level of 100).

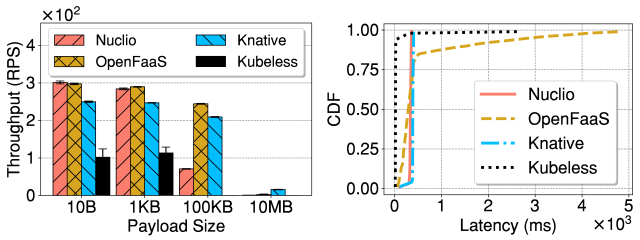


(a) Avg. throughput **(b)** Latency for concurrency of 100.

Figure 7. Throughput & latency of 'http-workload' function across different serverless platforms at different concurrency levels.

3.2.3 Variable Payload Size

For this experiment, in order to assess the data transfer overhead of serverless platforms, we scale the size of payload in the HTTP response and analyze the overheads and impact of assembling, packaging and transporting the HTTP response payload across different serverless platforms. In Fig. 8, we observe that Nuclio performs better for small payload sizes (*i.e.,* less than 1KB), while OpenFaaS and Knative perform better for large payloads.



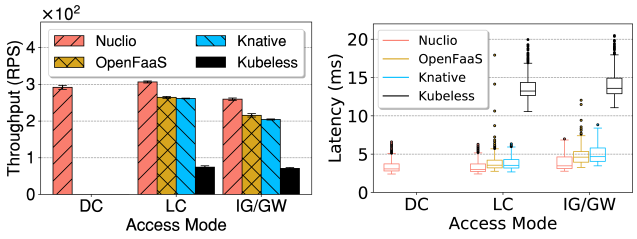
(a) Throughput in requests/second. **(b)** Latency for 1KB payload.

Figure 8. Throughput & latency of 'http-workload' function with different payload sizes for different serverless platforms.

3.2.4 Impact of different modes of exporting services

In order to avoid the added queuing latency, we run the http workload with `wrk` tool and limit the number of maximum concurrent (in-flight) requests to 1 and repeat the experiment 1000 times. Fig. 9 shows the impact on throughput and latency for three different modes of exporting and invoking the serverless functions. **LC** refers to local call, where the client and function pods reside on the same node in the Kubernetes cluster, and client invokes the function directly using the IP-address of the function pod. Nuclio has marginally better throughput and lower latency than Knative and OpenFaaS, while Kubeless suffers in both latency and throughput. **IG/GW** refers to exporting and invocation of serverless function through the ingress/API gateway components. This mode brings

down the throughput across all platforms, and also incurs (~ 1ms) additional latency than 'LC' mode. **DC** (direct call) approach is only supported by Nuclio, which exports the function pod using the nodeport service of Kubernetes. DC avoids the additional routing overheads in the worker node (netfilter rules that translate the packet destination, and forward the packets to the function pod).

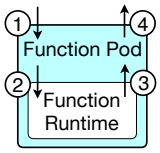


(a) Throughput in requests/second. **(b)** Latency in ms.

Figure 9. Throughput & latency for different methods of exporting the services on different serverless platforms.

3.2.5 Analysing the latency impact of serverless platforms

We analyze the delay overheads incurred in processing the 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 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. Time intervals between these timestamps are shown in Fig. 10. 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. Also, Kubeless (due to forking per request), incurs very high delay in forwarding the packet to the function instance. We also experimented with the 'http-function' and found the startup and response delay overheads to be same.



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

Figure 10. Latency breakdown (ms) parts of serverless execution.

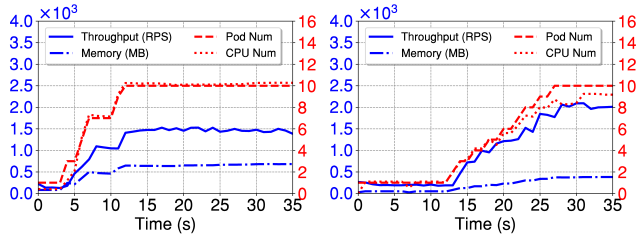
3.3 Auto-scaling

Auto-scaling capabilities exported by different serverless platforms vary. Here, we compare the auto-scaling features of Knative and OpenFaaS for both the rate-based and Kubernetes-based horizontal-pod-autoscaler (HPA) modes under different workload characteristics. For a fair comparison, we tune the auto-scaling related configuration parameters in both the platforms to have the same interval for the auto-scale triggers and factors for scaling functions.[11] We use the same python function as in §3.2.2. **Note:** In OpenFaaS, auto-scaling is based on the average rate of the incoming requests (RPS); in Knative, auto-scaling is based on the concurrency level observed per function instance. The subtle difference is that the average RPS value can be lower or higher than the observed concurrency depending on whether the time for processing a function invocation is higher or lower respectively. We will demonstrate the benefit and deficiency of both approaches.

[11] In Knative, we disable panic mode, and set the minScale and maxScale instances as 1 and 10, target to 10, max-scale-up-rate to 100, tick interval to 2s, and stable window to 10s, which ensures triggering auto-scale notifications on a 2s window and scaling to 1 or more instances at a time. Likewise, for OpenFaaS, we set scale-factor to 10 and configure the alert-notification window to 2s, and RPS threshold to 10. For HPA, we set CPU limits to 50.

3.3.1 Workload based auto-scaling

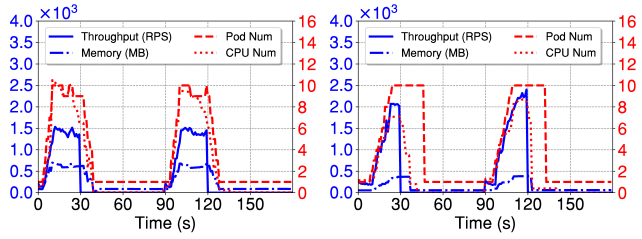
Steady workload: We use the `wrk` tool, set a steady rate for outstanding requests (concurrency of 100) and run the experiment for 60s. Also, to enforce proper traffic distribution across newly created pods, we force the OpenFaaS gateway to terminate and reestablish connections with the function pods. Periodically, every 2s, we monitor the number of pod instances, CPU and memory usage, and throughput. From Fig. 11, we observe that Knative scales multiple instances at a time to reach the max. (10) instances quickly (in 12s), while OpenFaaS just scales up one instance at a time, taking 26s to scale up to the max. (10) instances. Although, the CPU usage for the scaled instances looks identical, the memory pressure is higher for Knative. This stems from the difference in the python runtimes and overheads in the queue-proxy container component for Knative and of-watchdog components in OpenFaaS.



(a) Knative. **(b)** OpenFaaS.

Figure 11. Auto-scaling with steady workload.

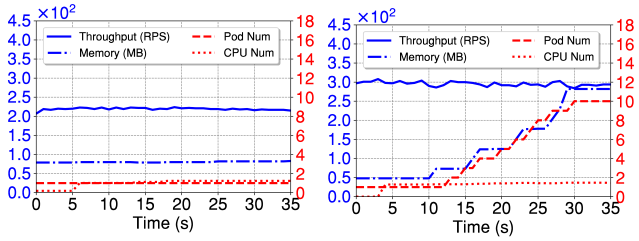
Bursty workload We also experimented by varying the http workload to have bursts of concurrent requests followed by a large idle period. Fig. 12 shows that, Knative is more responsive to bursts, and is able to scale quickly to a large number of instances, while OpenFaaS scales gradually and has lower average throughput.



(a) Knative. **(b)** OpenFaaS.

Figure 12. Auto-scaling with bursty workload.

Issues with auto-scaling in OpenFaaS and Knative: In another experiment, we use the same setup as in the steady workload experiment, but lowered the number of outstanding requests from 100 to 9. From Fig. 13, we observe that Knative fails to auto-scale and continues to operate with just 1 function pod instance, resulting in almost 7× lower (200 RPS) throughput compared to the earlier case (1500 RPS). Next, we revert to vanilla OpenFaaS (*i.e.,* as in github, and disable the workaround of resetting the connections to the function pods), and run the same steady workload experiment. The



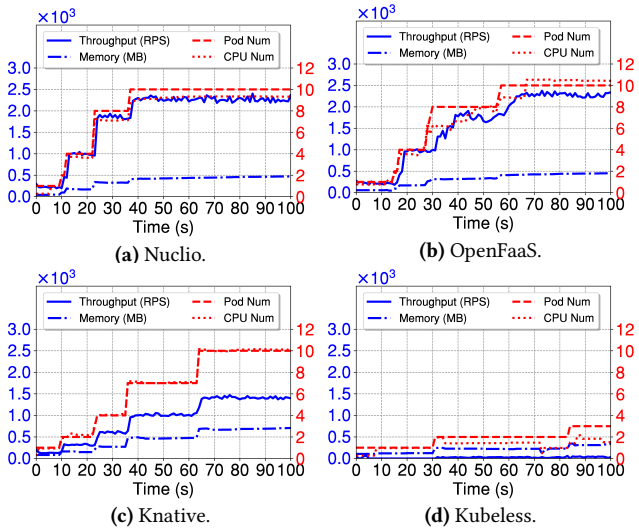
(a) Knative. **(b)** OpenFaaS.

Figure 13. Auto-scaling issues with Knative and OpenFaaS.

function pods get auto-scaled as before. But, the throughput shows no improvement. Also, note that with auto-scaling the memory usage increases, but CPU utilization remains steady. We found the issue to be due to incorrect traffic distribution. Due to the long running connections (setup by OpenFaaS gateway at the beginning with the first function pod), all the traffic is just routed only to the first function pod, while the remaining, newly scaled pods, do not receive any traffic.[12]

3.3.2 Resource based auto-scaling

We use the same setup (steady state), configure the cpu usage limit to 50%, and leverage Kubernetes HPA for auto-scaling. Note, the auto-scaling of function pods is governed by Kubernetes only. From Fig. 14, we can observe that, except for Kubeless, the auto-scaling behavior is same across all the platforms *i.e.,* auto-scaling tries to double the instances at each step until it reaches the maximum (10). However, the duration of each step depends on the CPU utilization factor, which in turn depends on the serverless platform specific components (event-listener, of-watchdog, queue-proxy). Nuclio, being relatively more CPU hungry is able to scale more rapidly (in 40s), than Knative and OpenFaaS. With Kubeless, the fork-per-request results in high latency, dropping of incoming requests that in turn results in low throughput and low CPU utilization. Thus, it results in poor auto-scaling as well.



(a) Nuclio. **(b)** OpenFaaS.

(c) Knative. **(d)** Kubeless.

Figure 14. HPA-based auto-scaling on steady workload.

4 Related Work

Serverless Platform comparison: In work [6, 11], the authors conducted several measurements on different cloud serverless platforms (AWS Lambda, Microsoft Azure, Google Cloud), and found the AWS to be better in terms of throughput, scalability, cold-start latency. The works [5, 12] investigate the different factors that influence the performance of AWS lambda, namely the impact of the choice of language of the function, memory footprint of the function, *etc.* Work [7] evaluates the performance of Fission, Kubeless and OpenFaaS serverless frameworks and characterizes the response time and the ratio of successfully completed requests for different loads. However the work fails to characterize the throughput of these platforms and accounts for the mean latency (response time) and successful responses at different load characteristics, which is debatable, without the proper consideration and configuration of the serverless platform specific configuration parameters, resulting in inaccurate results. In the most recent work [8], the authors quantitatively evaluate Apache OpenWhisk, OpenFaas, Kubeless, and Knative platforms. The results for Kubeless are similar, but for the other platforms, we feel the presented results are inaccurate. This could be due to the usage of Kubernetes. In contrast, our work focuses on discerning the architectural blocks that impact the performance of Kubernetes based open-source serverless platforms.

5 Summary

Through measurements, we explored different open-source serverless platforms and identified the key design considerations and their impact on performance and auto-scaling. We show that the interaction between the API Gateway/Ingress controller and the function pods, the overheads of this component and the way requests are queued influence baseline performance. Further, the 'RPS'-based and 'Concurrency'-based auto-scaling approaches by themselves are insufficient and need to evolve to properly meet workload demands, so that we can avoid maintaining a large number of instances active. **Acknowledgements**: This work was supported by US NSF grants CRI-1823270 and CNS-1763929, and grants from Hewlett Packard Enterprise Co., Futurewei Technologies Inc, and the National Key Research and Development Program of China under Grant 2018YFB1800100, 2018YFB1800500, 2018YFB1800800, and China Scholarship Council.

References

[1] 2018. wrk: a HTTP benchmarking tool. https://github.com/wg/wrk. [ONLINE].
[2] Sarah Allen and et al. 2018. CNCF Serverless Whitepaper. https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf. [ONLINE].
[3] Amazon. 2019. AWS Lambda. https://aws.amazon.com/lambda. [ONLINE].
[4] Brendan Burns and et al. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.
[5] Wes Lloyd and et al. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 159–169.
[6] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 405–410.
[7] S. K. Mohanty, G. Premsankar, and M. di Francesco. 2018. An Evaluation of Open Source Serverless Computing Frameworks. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 115–120.
[8] Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. 2019. An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge. In *2019 IEEE World Congress on Services (SERVICES)*, Vol. 2642. IEEE, 206–211.
[9] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *The magazine of USENIX & SAGE* 39, 6 (2014), 36–38.
[10] The New Stack. 2018. The New Stack Serverless Survey 2018. https://thenewstack.io/guide-to-serverless-technologies-free-ebook-on-the-new-stack/. [ONLINE].
[11] Liang Wang and et al. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 133–146.
[12] Cui Yan. 2017. How does language, memory and package size affect cold starts of AWS Lambda? https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76. [ONLINE].

[12] Bug raised: https://github.com/openfaas/faas/issues/1303.