Understanding Open Source Serverless Platforms: Design Considerations and Performance

Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, Dan Li
Open Source Serverless Platforms
Motivation and Goals

❖ To develop an understanding on the open source serverless platforms:
  ➢ Do measurements to understand the impact of key configuration parameters of different components (platform, gateway, controller and function)
❖ Evaluate and compare the performance of open source serverless platforms:
  ➢ Different workloads
  ➢ Different auto-scaling modes
Dependence on Kubernetes

Serverless Platforms
- Kubeless
- fission
- nuclio
- OpenFaas
- Apache OpenWhisk

Kubernetes
- Network Routing
- Traffic distribution
  Load-balancing
- Auto-scaling
- Orchestration & Management
- Logs, Debug and Monitoring
Service Exporting and Routing
Configure mapping rules for exporting service (Netfilter NAT rules)
Service Exporting and Routing

Configure mapping rules for exporting service (Netfilter NAT rules)

Encapsulate

Decapsulate

Public NodePort ⇧

Internal Pod_IP

Netfilter → Flannel Tun-Tap

Client
Service Exporting and Routing

Configure mapping rules for exporting service (Netfilter NAT rules)

Public NodePort ⇆ Internal Pod_IP

Encapsulate

Decapsulate

Route & Load balance
Service Exporting and Routing

Configure mapping rules for exporting service (Netfilter NAT rules)

Encapsulate

Public NodePort ↔ Internal Pod_IP

Get events and push to a worker

Execute functions

Route & Load balance

Decapsulate
Motivation and Goals

❖ To develop an understanding on the open source serverless platforms:
  ➢ Do measurements to understand the impact of key configuration parameters of different components (gateway, controller and function)

❖ Evaluate and compare the performance of open source serverless platforms:
  ➢ Different workloads
  ➢ Different auto-scaling modes

nuclio
OPENFAAS
K
Kubeless

v1.1.16
Gateway: v0.17.0
Faas-netes: v0.8.6
Faas-cli: v0.9.2

v0.8

v1.0.4
Experiment Setup

- **Topology**: Kubernetes cluster (1 master, 2 workers) on CloudLab
  - Hardware: Intel Xeon E5-2640 v4 @ 20 Hyperthread cores.
  - Operating System: Ubuntu 16.04.1 LTS
  - Kubernetes v1.16.1, Docker v18.09.2

- **Functions and Workload**: 
  - Python ‘Hello-world’ function
  - Python ‘HTTP’ function
  - Workload Generator: wrk

Nuclio

- Working model
Nuclio

- Working model

Invoke through IG
Nuclio

- Working model

Direct call to func pod (NodePort)

Invoke through IG
Nuclio

- Working model

1. Direct call to func pod (NodePort)
2. Invoke through IG
3. Multiple Workers (Processes)
Nuclio

- Salient parameter: the number of workers within one pod

(a) Throughput in requests/second.

(b) Latency in ms.
Nuclio

- Performance increases as the number of workers increases.

- Salient parameter: the number of workers within one pod

(a) Throughput in requests/second.

(b) Latency in ms.
Knative

- Working model
Knative

- Working model

Diagram:
- Client
- (Third party) Ingress Controller
- Function Pod
  - Queue-proxy Container
  - Function Container
- Multiple Workers (Threads)
Salient parameter: the number of workers within one pod

![Diagram of Knative architecture](image)

**Fig. Throughput of Knative.**  
**Fig. Throughput of Nuclio.**
Knative

- Performance improves, but relatively lower than Nuclio.
- Salient parameter: the number of workers within one pod

Fig. Throughput of Knative.  
Fig. Throughput of Nuclio.
OpenFaaS

- Working model

Client

OpenFaaS API Gateway

Function Pod

Of-Watchdog

Function Process
OpenFaaS

- Working model

Client

OpenFaaS API Gateway

Function Pod

Of-Watchdog

Function Process

One Worker
OpenFaaS

- Working model

Client

OpenFaaS API Gateway

Function Pod

Of-Watchdog

Function Process

Get events and invoke function

One Worker
OpenFaaS

- Working model

Multiple Modes for Of-Watchdog:
(1) Fork-per-request:
Cold start for every request;
(2) Pre-fork:
Start the function once and keep warm.

Get events and invoke function

One Worker
OpenFaaS

- Salient parameter: of-watchdog modes

- Throughput in requests/second.

- Latency in ms.
OpenFaaS

“Pre-fork” mode has much better performance than “Fork-per-request”.

- Salient parameter: of-watchdog modes
Kubeless

- Working model
Kubeless

- Working model

Kubeless only supports “Fork-per-request” mode.

(Fork-per-request)

One Worker

Client

(Third Party) Ingress Controller

Function Pod

Function Container

29
Kubeless

Fig. Throughput of Kubeless.

Fig. Throughput of OpenFaaS.
The performance of Kubeless is similar to that of OpenFaaS in “Fork-per-request” mode.

Fig. Throughput of Kubeless.

Fig. Throughput of OpenFaaS.
Motivation and Goals

❖ To develop an understanding on the open source serverless platforms:
  ➢ Describe how different components work
  ➢ Do measurements to understand the impact of key configuration parameters of different components (platform, gateway, controller and function)

❖ Evaluate and compare the performance of open source serverless platforms:
  ➢ Different workloads
  ➢ Different auto-scaling modes
Performance

Baseline: helloworld function (Return “hello”)

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

Baseline: helloworld function (Return “hello”)

- **Nuclio:**
  - No ingress controller
  - Bypass the queue of ingress controller
  - Highest throughput
Performance

Baseline: helloworld function (Return “hello”)

Nuclio:
- No ingress controller
  ⇒ Bypass the queue of ingress controller
  ⇒ Highest throughput

Kubeless:
- Fork-per-request
  ⇒ Lowest throughput
Performance

Baseline: helloworld function (Return “hello”)

- **Nuclio:**
  - No ingress controller
  - Bypass the queue of ingress controller
  - Highest throughput

- **Kubeless:**
  - Fork-per-request
  - Lowest throughput

Queuing Up ⇒ Long tail
Performance

Latency breakdown of helloworld function:

<table>
<thead>
<tr>
<th>Process</th>
<th>1→2</th>
<th>2→3</th>
<th>3→4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuclio</td>
<td>0.63</td>
<td>0.001</td>
<td>0.54</td>
</tr>
<tr>
<td>OpenFaaS</td>
<td>1.32</td>
<td>0.001</td>
<td>0.93</td>
</tr>
<tr>
<td>Knative</td>
<td>1.30</td>
<td>0.001</td>
<td>0.62</td>
</tr>
<tr>
<td>Kubeless</td>
<td>4.96</td>
<td>0.001</td>
<td>2.63</td>
</tr>
</tbody>
</table>

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

Latency breakdown of helloworld function:

Same Python Runtime ⇒ Same execution time

![Figure 10. Latency breakdown (ms) parts of serverless execution.](image)

<table>
<thead>
<tr>
<th>Process</th>
<th>1→2</th>
<th>2→3</th>
<th>3→4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuclio</td>
<td>0.63</td>
<td>0.001</td>
<td>0.54</td>
</tr>
<tr>
<td>OpenFaaS</td>
<td>1.32</td>
<td>0.001</td>
<td>0.93</td>
</tr>
<tr>
<td>Knative</td>
<td>1.30</td>
<td>0.001</td>
<td>0.62</td>
</tr>
<tr>
<td>Kubeless</td>
<td>4.96</td>
<td>0.001</td>
<td>2.63</td>
</tr>
</tbody>
</table>
Performance

Latency breakdown of helloworld function:

- **Same Python Runtime** ⇒ **Same execution time**

**Platform Specific**

<table>
<thead>
<tr>
<th>Process</th>
<th>1→2</th>
<th>2→3</th>
<th>3→4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuclio</td>
<td>0.63</td>
<td>0.001</td>
<td>0.54</td>
</tr>
<tr>
<td>OpenFaaS</td>
<td>1.32</td>
<td>0.001</td>
<td>0.93</td>
</tr>
<tr>
<td>Knative</td>
<td>1.30</td>
<td>0.001</td>
<td>0.62</td>
</tr>
<tr>
<td>Kubeless</td>
<td>4.96</td>
<td>0.001</td>
<td>2.63</td>
</tr>
</tbody>
</table>

**Figure 10.** Latency breakdown (ms) parts of serverless execution.
Latency breakdown of helloworld function:

- **Fork** - Per Request (Cold Start All the Time)

**Platform Specific**

Same Python Runtime $\Rightarrow$ Same execution time

<table>
<thead>
<tr>
<th>Process</th>
<th>1→2</th>
<th>2→3</th>
<th>3→4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuclio</td>
<td>0.63</td>
<td>0.001</td>
<td>0.54</td>
</tr>
<tr>
<td>OpenFaaS</td>
<td>1.32</td>
<td>0.001</td>
<td>0.93</td>
</tr>
<tr>
<td>Knative</td>
<td>1.30</td>
<td>0.001</td>
<td>0.62</td>
</tr>
<tr>
<td>Kubeless</td>
<td>4.96</td>
<td>0.001</td>
<td>2.63</td>
</tr>
</tbody>
</table>

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

HTTP Workload: fetch a web page (5 Byte) from a local server
Performance

HTTP Workload: fetch a web page (5 Byte) from a local server

Nuclio:
- No ingress controller
- Bypass the queue of ingress controller
- Highest throughput
Performance

Different modes of exporting services:
Performance

Different modes of exporting services:

Direct call to `func pod (NodePort)`
Performance

Different modes of exporting services:

Direct call to `func pod` (NodePort)

Invoke through IC/GW
Performance

Different modes of exporting services:

- Direct call to `func pod` (NodePort)
- Invoke through IC/GW

IC/GW: Overhead of Ingress Controller/API Gateway.
Performance: Auto-scaling

Resource-based auto-scaling:

Resource-based auto-scaling depends on Kubernetes HPA (Horizontal Pod Autoscaler)
Performance: Auto-scaling

Resource-based auto-scaling:

Resource-based auto-scaling depends on Kubernetes HPA (Horizontal Pod Autoscaler)

In spite of the same function and HPA, platform characteristics govern auto-scaling. (Different performance ⇒ Different resource utilization ⇒ Different auto-scaling rate)
Performance: Auto-scaling

Workload-based auto-scaling:

Concurrency-based

(a) Knative.

RPS-based

(b) OpenFaaS.
Performance: Auto-scaling

Workload-based auto-scaling:

Prometheus reacts slowly \Rightarrow \text{Slow scaling}

Concurrency-based

RPS-based
Performance: Auto-scaling

Issues about load balancing for OpenFaaS:

Fig. RPS-based auto-scaling in OpenFaaS
Performance: Auto-scaling

Issues about load balancing for OpenFaaS:

Behavior: Auto-scaling happens but NO performance improvement!

No improvement
Auto-scale

Fig. RPS-based auto-scaling in OpenFaaS
Performance: Auto-scaling

Issues about load balancing for OpenFaaS:

Behavior: Auto-scaling happens but NO performance improvement!

Load-balancing Issue!
If client enables keep-alive, OpenFaaS does not set up connections with newly created function pods, which hinders performance improvement.

Fig. RPS-based auto-scaling in OpenFaaS

No improvement

Auto-scale
Performance: Auto-scaling

Issues about Concurrent-based auto-scaling:

Traffic: Conc=9, RPS=400
Configuration: Conc_Threshold=10

Fig. Conc-based auto-scaling in Knative
Performance: Auto-scaling

Issues about Concurrent-based auto-scaling:

Traffic: Conc=9, RPS=400
Configuration: Conc_Threshold=10

Behavior: No Auto-scaling!
No able to scale to 400 RPS (Actual RPS=\sim 220)

Fig. Conc-based auto-scaling in Knative
Performance: Auto-scaling

Issues about Concurrent-based auto-scaling:

Traffic: Conc=9, RPS=400
Configuration: Conc_Threshold=10

Behavior: No Auto-scaling!
No able to scale to 400 RPS (Actual RPS=\sim220)

Misconfiguration inhibits auto-scaling.
(Conc. does not exceed threshold. ⇒ No auto-scaling with workload of low concurrency but high RPS.)

Fig. Conc-based auto-scaling in Knative
Key Observations

❖ Function processing:
  ➢ Multiple workers within one function pod contribute to performance improvement.
  ➢ Pre-fork mode (warm worker) increases the throughput and reduces the latency.

❖ Load balancing:
  ➢ Plays an important role in the performance and scalability.
  ➢ Coupling routing with load balancing can adversely affect the performance -- Needs greater attention!
Key Observations

❖ Autoscaling:
  ➢ For resource-based auto-scaling, in spite of the same function and HPA, platform characteristics govern auto-scaling.
  ➢ Misconfiguration of auto-scaling rules can severely degrade the performance and system utilization.
 ➢ Current Auto-scaling approaches are based only on the total processed requests, while the dropped requests are missed out. -- Incoming request rate needs to be accounted for.
Backup Slides
Load balancing:
- Improper load balancing results in poor performance improvement -- Needs greater attention!

Autoscaling:
- Misconfiguration of auto-scaling rules can severely degrade the performance and system utilization.
- Current Auto-scaling approaches are based only on the total processed requests, while the dropped requests are missed out. -- Incoming request rate needs to be accounted for.
Motivation

❖ To understand how the serverless platforms work?

❖ What is the impact of configuration parameters?

❖ What is the performance of serverless platforms?

❖ What is the behavior of auto-scaling?
Thank you!
Performance

HTTP Workload: fetch a web page of different sizes

(a) Throughput in requests/second.
(b) Latency for 1KB payload.
Performance: Auto-scaling

Workload-based auto-scaling: bursty workload

Concurrency-based

RPS-based

Prometheus ⇒ React slowly

(a) Knative.

(b) OpenFaaS.

Auto-scaling with bursty workload
What is serverless?

Build and run applications without thinking about servers
Serverless Computing: The New Hotness