Is Function-as-a-Service (FaaS) a Good Fit for Latency-critical Services

Haoran Qiu¹, Saurabh Jha¹, Subho Banerjee¹, Archit Patke¹, Chen Wang², Hubertus Franke²
Zbigniew Kalbarczyk¹, Ravishakar Iyer¹

¹ University of Illinois, Urbana-Champaign ² IBM Research

Paper link: https://www.serverlesscomputing.org/wosc7/papers/p1
Seventh International Workshop on Serverless Computing (WoSC7) 2021
Traditional vs. FaaS – An Example

Application Owner

Develop

Pet Store Application

Pet Store Server

Virtual Machine / Physical Machine

Deploy

End User

API Gateway

Authentication Function

Search-Pet Function

Add/Delete-Pet Function

Database

Application Owner

Develop

Register to FaaS Platform

FaaS Provider

Provision & Manage

Database

Provision

Manage

Traditional

FaaS
Latency-critical Services

- Online Navigation
- Social Network
- Web Mail Service
- Machine Translation
Latency-critical Services

- Latency-critical services are typically user-facing and operate with strict service-level objectives (SLOs) on the end-to-end latency, especially the tail latency (e.g., 99th percentile of the requests returned to users < 100ms).

- Question: Is FaaS a good fit for latency-critical services?

  - Lower latency!
  - Higher utilization, Higher profit!

  Customers

  FaaS Provider
Resource Granularity in Workload Consolidation Policies

• We tune the **memory limit** of each container as FaaS platform allocates other type of resources proportionally to memory limits

• Resource granularities are discrete points on a spectrum

---

**Fine-grained (FG) Resource Allocation**
- Common in FaaS
- **Min:** 128MB
- **Min step size:** 1MB

**Coarse-grained (CG) Resource Allocation**
- Common in PaaS/IaaS
- **Min:** 512MB
- **Min step size:** 512MB

---

**Memory Limit Spectrum**

- **OpenWhisk, OpenFaaS, Knative (FaaS)**
  - **Min:** 4MB
  - **Min step size:** 1MB

- **AWS Lambda, Google Cloud Run, Google Cloud Functions (FaaS)**
  - **Min:** 128MB
  - **Min step size:** 1MB

- **Google Compute Engine (IaaS)**
  - **Min:** 256MB
  - **Min step size:** 256MB

- **Google GKE (CaaS)**
  - **Min:** 512MB
  - **Min step size:** 512MB

- **Heroku Platform, AWS EKS (PaaS)**
  - **Min:** 512MB
  - **Min step size:** 512MB

- **AWS ECS (CaaS), AWS EC2 (IaaS)**
  - **Min:** 512MB
  - **Min step size:** 512MB

---

Seventh International Workshop on Serverless Computing (WoSC7) 2021
Goal and Key Findings

• What is the trade-off among power consumption, CPU utilization, and end-to-end latency in the decision-making of choosing a workload consolidation policy?
  • Increasing resource granularity (e.g., increasing a container’s allocated memory limit from 128 MB to 256 MB):
    • Reduces tail latency by up to 2x,
    • Consumes up to 1.75× more power,
    • Reduces CPU utilization by up to 59%

• How is the performance variation affected by fine-grained workload consolidation?
• How do different workload consolidation policies affect the breakdown percentages of different phases in the end-to-end latency?
Goal and Key Findings

• What is the trade-off among power consumption, CPU utilization, and end-to-end latency in the decision-making of choosing a workload consolidation policy?
  • Increasing resource granularity (e.g., increasing a container’s allocated memory limit from 128 MB to 256 MB):
    • Reduces tail latency by up to 2x,
    • Consumes up to 1.75× more power,
    • Reduces CPU utilization by up to 59%

• How is the performance variation affected by fine-grained workload consolidation?
  • Shared resource contention leads to tail-latency increase of up to 32.6x, 28.9x, and 4.4x for CPU, memory, and LLC sensitive workloads
    • With state-of-the-art resource partitioning, tail-latency increase becomes 8.3x, 21.5x, and 2.3x

• How do different workload consolidation policies affect the breakdown percentages of different phases in the end-to-end latency?
Goal and Key Findings

• What is the trade-off among power consumption, CPU utilization, and end-to-end latency in the decision-making of choosing a workload consolidation policy?
  • Increasing resource granularity (e.g., increasing a container’s allocated memory limit from 128 MB to 256 MB):
    • Reduces tail latency by up to 2x,
    • Consumes up to $1.75 \times$ more power,
    • Reduces CPU utilization by up to 59%

• How is the performance variation affected by fine-grained workload consolidation?

• How do different workload consolidation policies affect the breakdown percentages of different phases in the end-to-end latency?
  • Increasing the horizontal concurrency (i.e., number of containers) from 2 to 12 on a single server via decreasing resource granularity:
    • Reduces tail wait time by 49.5x but increases tail init time by 1.3x and increases tail execution time by 15.6x
    • End-to-end latency breakdown varies with concurrency and workloads
Latency-Utilization-Power Trade-off

- Lower latency
  - Customers

- Higher utilization, Lower power consumption, Higher latency
  - FaaaS Provider

- Preferable utilization
- SLO latency
- PROVIDER PREFER:
  - Rate = 2

- CUSTOMER WITH LC SERVICES PREFER:
  - Rate = 2
Latency-Utilization-Power Trade-off

Slo latency

Provider prefers

Customer with LC Services prefers

Slo latency

Higher utilization, lower power consumption, higher latency

Lower latency

Customers

Preferrable utilization

Faas Provider
Latency-Utilization-Power Trade-off

- **Lower latency**
  - Customers

- **Higher utilization, Lower power consumption, Higher latency**
  - FaaS Provider

SLO latency

Preferrable utilization
Latency-Utilization-Power Trade-off

- Lower latency
- Higher utilization, Lower power consumption, Higher latency

SLO latency

Preferrable utilization
Latency-Utilization-Power Trade-off

- Lower latency
- Higher utilization, Lower power consumption, Higher latency
- SLO latency
- Preferrable utilization

Customers

FaaS Provider
Latency-Utilization-Power Trade-off

**Graph:**
- **Axes:**
  - Y-axis: Power Consumption (W)
  - X-axis: Horizontal Concurrency
- **Legend:**
  - Rate = 2
  - Rate = 3
  - Rate = 4
  - Rate = 5
  - Rate = 6
  - Fine-grained
  - Coarse-grained

**Diagram:**
- **Worker Node #CG1:**
  - Invoker
  - Controller
  - Docker
- **Worker Node #CG2:**
  - Invoker
  - Controller
  - Docker
- **Worker Node #FG:**
  - Invoker
  - Controller
  - Docker
[Implication] An FG policy leads to lower operation costs (up to 1.75× less) and better server utilization efficiency (up to 59% higher), while a CG policy offers the customers lower end-to-end latency (up to 2× less).

The conflicting goals of the two parties raise questions,
• On the pricing model: how to balance the needs of both parties?
• On the provider-customer interface: how should resource and performance needs be conveyed?
Thank you!

Check out our paper for more details: https://www.serverlesscomputing.org/wosc7/papers/p1

Code available at: https://github.com/James-QiuHaoran/serverless-wosc7
Backup Slides
Background: Serverless Function-as-a-Service (FaaS)

- Serverless computing
  - Cloud provider allocates and scales compute resources
  - Customers are charged for the compute resources used

- Function-as-a-Service (FaaS)
  - Customers writes code that only tackles application logic; uploads it to FaaS platform
  - No need to configure/manage the provisioning and maintenance of the resources
  - E.g., Google Cloud Functions, AWS Lambda, IBM Cloud Functions, Azure Functions

Focus of this paper
System Stack Management – Traditional vs. FaaS

• In traditional cloud computing paradigms, customers **configure and pay** for the cloud resources that they requested
  • E.g., the number of cores and amount of memory for a virtual machine
• Customers tend to **overprovision** compute resources to meet application end-to-end performance goals
• Operating system (VM) is the scale of unit
System Stack Management – Traditional vs. FaaS

- FaaS frees application developers from infrastructure management
  - E.g., resource provisioning, scaling
- Customers are charged by the compute resource usage during the execution time (no expense for idle times)
- FaaS provider creates containers for a function, scales the number of containers, and co-locates multiple containers on the same server (i.e., workload consolidation)
  - At the cost of higher end-to-end function request latencies (up to 2x from our evaluation results)
Experimental Setup Overview

• Measurements from the execution of 2 widely used FaaS benchmark suites
  • ServerlessBench, FaaS-Profiler
• Benchmarks running on an open-sourced FaaS platform -- OpenWhisk
• Deployed on IBM Cloud with 1 master node and 4 worker nodes
Concept Overview

Warm-start
- Request Received
- Invoker Found
- /run
- END
- Wait Time
- Exec Time

Cold-start
- Request Received
- Image/Code Pull
- /init
- /run
- END
- Init Time
- Exec Time

Container Provisioning & Initialization
Queueing
Latency Variation

Both on 1 node

FG on 1 node

CG scales to 2 nodes
Latency Variation

Both on 1 node

FG on 1 node

CG scales to 2 nodes

Wait time ↓ & Exec time ↑

Exec time ↑

Wait time ↓
Latency Variation

Both on 1 node

FG on 1 node
CG scales to 2 nodes

Wait time ↓ & Exec time ↑

Exec time ↑
Wait time ↓

[Implication] Compared to FG policies, a CG policy scales out containers on a greater number of servers, resulting in less resource contention and thus up to 67% lower end-to-end latency.
Latency-Utilization-Power Trade-off

![Latency-Utilization-Power Trade-off Diagram](image-url)

Table 1: Latency-Utilization-Power Trade-off Values

<table>
<thead>
<tr>
<th>FG / CG in Memory Capacity (MB)</th>
<th>Horizontal Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>128/256</td>
<td>12.5%</td>
</tr>
<tr>
<td>128/512</td>
<td>10.7%</td>
</tr>
<tr>
<td>128/768</td>
<td>11.8%</td>
</tr>
<tr>
<td>128/1024</td>
<td>12.2%</td>
</tr>
<tr>
<td>160/256</td>
<td>10.6%</td>
</tr>
<tr>
<td>160/512</td>
<td>9.9%</td>
</tr>
<tr>
<td>160/768</td>
<td>6.5%</td>
</tr>
<tr>
<td>160/1024</td>
<td>12.1%</td>
</tr>
<tr>
<td>192/256</td>
<td>11.4%</td>
</tr>
<tr>
<td>192/512</td>
<td>8.2%</td>
</tr>
<tr>
<td>192/768</td>
<td>4.0%</td>
</tr>
<tr>
<td>192/1024</td>
<td>5.3%</td>
</tr>
</tbody>
</table>
Performance Interference

**CPU Time Contention**

| Base64 (Avg) | 166.8% | 1497.7% | 1712.7% |
| Base64 (Tail) | 182.0% | 1451.4% | 1656.4% |
| Primes (Avg) | 114.9% | 184.3% | 250.0% |
| Primes (Tail) | 131.8% | 272.4% | 333.2% |
| Markdown2HTML (Avg) | 404.5% | 1120.0% | 1299.8% |
| Markdown2HTML (Tail) | 410.8% | 1174.9% | 1383.9% |
| Sentiment (Avg) | 128.7% | 239.0% | 349.3% |
| Sentiment (Tail) | 158.0% | 321.7% | 427.2% |
| Image-Resize (Avg) | 984.1% | 2920.5% | 3942.5% |
| Image-Resize (Tail) | 828.2% | 2883.3% | 3258.5% |

**Memory Bandwidth Contention**

| | 131.3% | 175.8% | 261.0% | 521.3% | 663.9% |
| Base64 (Avg) | 200.0% | 223.8% | 262.4% | 517.2% | 671.7% |
| Base64 (Tail) | 105.2% | 108.8% | 114.1% | 126.2% | 139.5% |
| Primes (Avg) | 136.6% | 148.2% | 153.5% | 167.0% | 177.8% |
| Primes (Tail) | 100.1% | 103.7% | 492.1% | 525.9% | 806.7% |
| Markdown2HTML (Avg) | 121.8% | 125.9% | 463.8% | 549.6% | 828.0% |
| Markdown2HTML (Tail) | 112.9% | 180.4% | 183.4% | 193.0% | 204.6% |
| Sentiment (Avg) | 102.9% | 115.8% | 122.2% | 135.7% | 164.2% |
| Sentiment (Tail) | 133.9% | 134.1% | 898.6% | 1743.7% | 2400.6% |
| Image-Resize (Avg) | 125.7% | 127.9% | 1241.5% | 1273.4% | 2151.5% |
| Image-Resize (Tail) | 125.7% | 127.9% | 1241.5% | 1273.4% | 2151.5% |

**LLC Contention**

| | 101.0% | 101.0% | 103.2% | 106.8% |
| Base64 (Avg) | 101.0% | 103.0% | 104.1% | 107.0% |
| Base64 (Tail) | 100.7% | 101.0% | 102.2% | 104.5% |
| Primes (Avg) | 100.2% | 100.3% | 101.4% | 105.6% |
| Primes (Tail) | 106.9% | 136.6% | 175.4% | 230.6% |
| Markdown2HTML (Avg) | 101.3% | 126.6% | 164.2% | 229.8% |
| Markdown2HTML (Tail) | 100.7% | 101.4% | 103.7% | 123.7% |
| Sentiment (Avg) | 100.9% | 101.5% | 101.5% | 128.8% |
| Sentiment (Tail) | 103.1% | 106.1% | 111.6% | 121.8% |
| Image-Resize (Avg) | 103.7% | 112.4% | 114.3% | 125.6% |
[Implication]

- Performance isolation should be carefully assessed to prevent SLO violations due to resource sharing.
- However, when thousands of function containers are consolidated on a single server, state-of-the-art resource partitioning fails to mitigate the performance interference, still with up to $8.3 \times$, $21.5 \times$, and $2.3 \times$ increase in end-to-end tail latencies for CPU, memory, and LLC sensitive workloads.
End-to-end Latency Breakdown
[Implication] The three-phase breakdown of end-to-end latency varies with the concurrency-to-arrival-rate ratio. Increasing the concurrency from 2 to 12:
• Reduces the tail wait time by 49.5× from 1820 ms
• Increases tail initialization time by 1.3× from 409 ms
• Increases tail execution time by 15.6× from 484 ms