Characterizing Serverless Systems

Mohammad Shahrad

mshahrad@ece.ubc.ca
@MShahrad

The University of British Columbia

UBC Cloud Infrastructure Research for Reliability, Usability, and Scalability

January 2022
Some men were exhibiting an elephant in a dark room, and many people collected to see it. But as the place was too dark to permit them to see the elephant, they all felt it with their hands, to gain an idea of what it was like. One felt its trunk and declared that the beast resembled a water-pipe; another felt its ear and said it must be a large fan; another its leg and thought it must be a pillar; another felt its back and declared the beast must be like a great throne. According to the part which each felt, he gave a different description of the animal.
Characterization

1. All aspects
2. Meticulous
3. Realistic
4. Reproduceable

Serverless
Production Serverless Systems

External Characterization → APIs ← Internal Characterization

Full Characterization

Local Deployments
External Characterization

A great example of external characterization of serverless systems:

**Peeking Behind the Curtains of Serverless Platforms**

Liang Wang\(^1\), Mengyuan Li\(^2\), Yinqian Zhang\(^2\), Thomas Ristenpart\(^3\), Michael Swift\(^1\)

\(^1\)UW-Madison, \(^2\)Ohio State University, \(^3\)Cornell Tech

USENIX ATC 2018

Many valuable comparative data on:
- Cold start latency
- Instance lifetime
- …
Limitations of External Characterization

Usage patterns are unknown:
• Impossible to pick a set of benchmarks that represent real workloads.
• Even you have a representative workload, what is the invocation pattern?

The internals of the system are unknown:
• E.g.: where is the cold start time coming from?
  (retrieving the image, starting the language runtime, etc.)

Noise:
• Network delays
• Asynchronous clocks for servers logging the events
Limitations of Internal Characterization

- You are at a provider’s mercy to 1) do this and 2) release the data! 😞

- Certain aspects of the characterization can be specific to that provider:
  → data from multiple providers can help
Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider

Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini *

Microsoft Azure and Microsoft Research

• First characterization of production serverless workloads
  • Many new insights
  • Released production traces of Azure Functions:
    • https://github.com/Azure/AzurePublicDataset

• A new adaptive serverless management scheme
  • New angle: going after eliminating cold starts instead of reducing cold start overhead
  • Improves the user experience, reduces the underlying resource usage
  • Deployed in production
Invocations per Application

This graph is from a representative subset of the workload. See paper for details.
Invocations per Application

This graph is from a representative subset of the workload. See paper for details.
Invocations per Application

Average Interval Between Invocations

This graph is from a representative subset of the workload. See paper for details.
Invocations per Application

This graph is from a representative subset of the workload. See paper for details.
Invocations per Application

This graph is from a representative subset of the workload. See paper for details.
Invocations per Application

This graph is from a representative subset of the workload. See paper for details.
This graph is from a representative subset of the workload. See paper for details.
Who should you design/optimize the system for?

18% of applications that constitute 99.6% of invocations

- Can lead to immediate cost reductions for the provider.
- Keeps the big users happy and prevents them from migrating to competitors.

Or

82% of applications that constitute 0.4% of invocations

- Can incentivize more users to shift to serverless. Potentially higher long-term profit.
- Seems reasonable when user valuation of service/QoS is unknown.

Mixing both?
Average Invocations Do Not Tell the Entire Story
Performant Serverless
- Short Execution
- Low Cold Start Overhead
- Fewer Cold Starts
- Minimum Wait Time

Efficient Serverless
- High Co-tenancy
- Minimal Resource Wastage
- High Cluster Utilization
It is all about trade-offs!

Performant Serverless
- Short Execution
- Low Cold Start
- Fewer Cold Starts
- Minimum Wait Time

Efficient Serverless
- High Co-tenancy
- Minimal Resource Wastage
- High Cluster Utilization

Conflicting Goals
Cold Starts and Resource Wastage
Cold Starts and Resource Wastage

Wasted Memory

Keeping functions in memory indefinitely.

Cold Starts

Removing function instance from memory after invocation.
Removing function instance from memory after invocation.

$O(\text{execution time}) = O(\text{cold start overhead})$

Cold starts cannot be neglected.

Keeping functions in memory indefinitely.

Memory usage not negligible.

Can’t be kept in memory forever.
What do serverless providers do?

Mikhail Shilkov, Cold Starts in Serverless Functions, https://mikhail.io/serverless/coldstarts/
Fixed Keep-Alive Policy

Results from simulation of the entire workload for a week.
A Histogram Policy To Learn Idle Times

- Pre-warm
- Keep-alive

Frequency

Minute-long bins

Limited number of bins (e.g., 240 bins for 4-hours)

Idle Time (IT)

5th percentile

99th percentile

Limited number of bins (e.g., 240 bins for 4-hours)
We can afford to run complex predictors given the low arrival rate. A histogram might be too wasteful.
**Decision Tree for the Hybrid Histogram Policy**

- **ARIMA:** Autoregressive Integrated Moving Average

- **New invocation** → **Update app’s IT distribution** → **Too many OOB ITs**
  - No: **Pattern Significant**
  - Yes: **Use IT distribution (histogram)**

- **Pattern Significant**
  - Yes: **Use IT distribution (histogram)**
  - No: **Be conservative (standard keep-alive)**

- **Too many OOB ITs**
  - Yes: **Time-series forecast (ARIMA)**
  - No: **Be conservative (standard keep-alive)**
More Optimal Pareto Frontier
Container memory reduction: 15.6%
Average exec time reduction: 32.5%
99th-percentile exec time reduction: 82.4%
Latency overhead: < 1ms (~800 µs)

Production possible thanks to the dedication of Íñigo Goiri, Gohar Chaudhry, and the Azure Functions team.
Many viable options available. Full control and visibility.
• Open-sourced industry-grade (IBM Cloud Functions)
• Functions run in containers
• Functions can be in Python, Node.js, Scala, Java, Go, Ruby, Swift, PHP, .Net, and Rust
Are “serverless” servers utilized efficiently?

- No Benchmark
- Let’s gather some!
- Too Complex of a System
- Let’s try to understand it first!

Findings presented to and well received by the OpenWhisk team.

No Clear Testing & Profiling Methodology
Let’s build one!
Building a Testing/Profiling Tool (FaaSProfiler)

https://github.com/PrincetonUniversity/faas-profiler

Shahrad et al., Architectural Implications of Function-as-a-Service Computing [MICRO ’19]
Benchmarks and Test Setup

Benchmarks:

• 5 representative applications
• 28 Python microbenchmarks

Test server:

• Intel Xeon E5-2620 v4
• 8-cores, 16-threads
• 20MB Last-Level Cache
• 16GB 2133MHz DDR4 (single-channel)

<table>
<thead>
<tr>
<th>FaaS Benchmark</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>autocomplete</td>
<td>NodeJS</td>
</tr>
<tr>
<td>markdown-to-HTML</td>
<td>Python</td>
</tr>
<tr>
<td>img-resize</td>
<td>NodeJS</td>
</tr>
<tr>
<td>sentiment-analysis</td>
<td>Python</td>
</tr>
<tr>
<td>ocr-img</td>
<td>NodeJS + binary</td>
</tr>
</tbody>
</table>

Representative applications gathered and pruned by Jonathan Balkind.
Server Capacity & Latency Modes

- Over-invoked
- Capacity
- Balanced
- Under-invoked

Containers paused repeatedly

- Rate > Capacity
- Rate = Capacity
- Rate < Capacity
- Rate << Capacity

Shahrad et al., Architectural Implications of Function-as-a-Service Computing [MICRO ’19]
Latency Breakdown

1. Initialization Time
2. Wait Time in the Queue
3. Execution Time

Non-trivial System Overhead
Architectural Limitations

Last-Level Cache

Branch Prediction

Memory Bandwidth
We observed low LLC requirement.

Used Intel Cache Allocation Technology (CAT).

Shahrad et al., Architectural Implications of Function-as-a-Service Computing [MICRO ’19]
Decreasing LLC Requirement, a Common Trend

• Scale out workloads [Ferdman et al., ASPLOS ‘12]
• Latency-critical cloud workloads [Chen et al., ASPLOS ‘19]
• Microservices [Gan et al., ASPLOS ‘19]
• FaaS [Shahrad et al., MICRO ‘19]

Opportunity

Aggressive co-location with cache-sensitive workloads.

Dedicating more silicon to logic than SRAM.
MPKI does not vary with invocation rate if containers kept alive.

MPKI: Misses per Kilo Instructions

Functions have a distinct behavior.
Longer execution helps with branch misses.

Shorter functions have higher MPKI. [\sim 20x variations]

Simulations revealed the reason.

GShare simulations performed by Jonathan Balkind.

Shahrad et al., Architectural Implications of Function-as-a-Service Computing [MICRO ’19]
Questioning Conventional Microarchitectural Expectations

• Conventional expectation: programs run for long enough to train the predictors.
• Short deeply-virtualized functions are not a good fit to this model.

Opportunity:

Revised branch predictors for:

• Retaining prediction states at the container- or application-level
  • Faster training
Various demands make it hard to co-locate.

- Heavy payload [O(1MB)]
- Short execution time
- Light payload [O(10B)]
- Very short execution time
- Medium payload [O(100KB)]
- Long execution time
Serverless in the Wild:
• Traces enable new wave of academic research
• New angle (cold starts instead of cold start overhead)

Architectural Implications of FaaS:
• Got many architects excited about this new domain
• Provided a new tool to accelerate research
Closing the Loop

• We need more data from real serverless systems.

• Careful characterization is key.
  [data-center-level, cluster-level, system-level, server-level]

• Any characterization study has its limitations.

• Complex systems make it easy to leave efficiency behind the door.