The Dawn of the Cloud Computer

Rodric @Rabbah

Fifth International Workshop on Serverless Computing
WoSC 2019
5 years ago
Amazon announced…
instantly reactive functions

```javascript
let main = () => ({
  msg: "Hello World"
})
```
“example”

> let hello = ...
> open bit.ly/hello-fn
no Server logic

```javascript
let main = () => {
  msg: "Hello World"
}

server.route('/hello',

server.listen(port)
```
no Server at all

global server.

```javascript
server.route('/hello',

let main = () => ({
  msg: "Hello World"
})
)

server.listen(port)
```

@rabbah - Workshop on Serverless Computing 2019
highly concurrent by default

Running 10s test @ https://apigcp.nimbella.io/api/v1/web/rabbahgm-rg0c4xagzcl/default/hello.json
10 threads and 10 connections

<table>
<thead>
<tr>
<th>Thread Stats</th>
<th>Avg</th>
<th>Stdev</th>
<th>Max</th>
<th>+/- Stdev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>43.86ms</td>
<td>30.24ms</td>
<td>284.36ms</td>
<td>91.30%</td>
</tr>
<tr>
<td>Req/Sec</td>
<td>25.36</td>
<td>7.86</td>
<td>40.00</td>
<td>83.12%</td>
</tr>
</tbody>
</table>

2482 requests in 10.06s, 1.19MB read
Requests/sec: 246.64
Transfer/sec: 120.90KB
Serverless.
Serverless benefits
$10^3$ concurrency in seconds

$10^6$ operations < $0.25$
Outsourcing Everyday Jobs to Thousands of Cloud Functions with gg

Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Alex Ozdemir, Shuvo Chatterjee, Matei Zaharia, Christos Kozyrakis, and Keith Winston

Occupying the Cloud: Distributed Computing for the 99%
Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, Benjamin Recht
University of California, Berkeley
{jonas, qifan, shivaram, istoica, brecht}@eecs.berkeley.edu

ABSTRACT
Distributed computing remains inaccessible to a large number of users, in spite of many open source platforms and extensive commercial offerings. While distributed computation frameworks have target on-premise installations at large scale. On commercial cloud platforms, a novice user confronts a dizzying array of potential decisions: one must decide in advance whether to instance type, cluster size, pricing model, programming model, and task granularity. Such choices are particularly necessary when selecting the

PyWren: Real-time Elastic Execution
PyWren is a system we built to enable incredibly scalable execution on the cloud using AWS Lambda (and other "serverless" frameworks) mean it -- you can nearly-instantly run your code on literally thousand overhead, all billed in 100ms-increments.

numpypwren: Serverless Linear Algebra
Vaishaal Shankar¹, Karl Krauth¹, Qifan Pu¹,
Eric Jonas¹, Shivaram Venkataraman², Ion Stoica¹, Benjamin Recht¹, and Jonathan Ragan-Kelley¹
¹UC Berkeley
²UW Madison

Abstract
Linear algebra operations are widely used in scientific computing and machine learning applications. However, it is challenging for scientists and data analysts to run linear algebra at scales beyond a single machine. Traditional approaches either require access to supercomputing clusters, or impose configuration and cluster management challenges. In this paper we show how the disaggregation of "compute and manage" operations in so-called "serverless" computing frameworks. This allows us to leverage the large-scale compute capacity of a public cloud platform to execute these operations at scale.
It is not all academic.
$100M+ investor backed serverless startups in 2018
The cloud computing landscape is changing.
The cloud computing landscape is changing. Has changed.
Increasing automation & abstraction are engrained in the history of computing.
Increasing **automation & abstraction** are engrained in the **history** of computing.
2015
2 Billion Lambdas / day

2019
2+ Trillion / month
Serverless is inevitable.
AWS re:Invent 2019
AWS re:Invent Livestream Dr. Werner Vogels

Ability to innovate

COST OF COMPUTE
30%

COST TO BUILD
30%

20x

DEPLOYMENT FREQUENCY
My Serverless Conjecture

The number of servers managed by an organization will decrease in half every 2 years.
there are limits... of course
Function Isolation

```javascript
function main() {
  return {
    msg: "Hello World"
  }
}
```
Serverless Elasticity

resource isolation and provisioning

containers

500ms
Container Lifecycle

create container → initialize container → run container → delete container
function execution delay

queue execution

function concurrency

vendor costs (resources)

execute instantly
Serverless Tensions

scale infinitely vs. control costs
execute instantly finite resources

bit.ly/serverless-contract
Serverless Contract

\( X \% \) of the time the function will start to execute in \( Y \) milliseconds

bit.ly/serverless-contract
Serverless Contract

Arrival Rate
A events / seconds

Drain Rate
D functions / seconds
Serverless Contract

\[ A < D : \text{queuing latency} \approx 0 \]

*The system is over-provisioned.*
Serverless Contract

A \approx D: \text{ queuing latency } \approx 0

Balanced but difficult to achieve with dynamic load.
Serverless Contract

$A > D$: queuing latency $\propto$ mismatch

The system is under-provisioned.
Cold Starts
## Wait Time

<table>
<thead>
<tr>
<th>Activation List</th>
<th>Execution Time</th>
<th>Queueing Delays</th>
<th>Container Initialization</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>9998da082 hello</td>
<td>4ms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25ba9eba6 hello</td>
<td>4ms</td>
<td></td>
<td></td>
<td>-26ms</td>
</tr>
<tr>
<td>729ae2d48 hello</td>
<td>5ms</td>
<td></td>
<td></td>
<td>+32ms</td>
</tr>
<tr>
<td>5b368ba47 hello</td>
<td>2ms</td>
<td></td>
<td></td>
<td>-53ms</td>
</tr>
<tr>
<td>10e90d350 hello</td>
<td>3ms</td>
<td></td>
<td></td>
<td>-9ms</td>
</tr>
<tr>
<td>c6af1d472 hello</td>
<td>105ms</td>
<td></td>
<td></td>
<td>-77ms</td>
</tr>
<tr>
<td>ac248c0ba hello</td>
<td>8ms</td>
<td></td>
<td></td>
<td>+68ms</td>
</tr>
<tr>
<td>fb0d6ad8f hello</td>
<td>9ms</td>
<td></td>
<td></td>
<td>+8ms</td>
</tr>
<tr>
<td>c9ba56333 hello</td>
<td>3ms</td>
<td></td>
<td></td>
<td>-48ms</td>
</tr>
<tr>
<td>60b192206 hello</td>
<td>7ms</td>
<td></td>
<td></td>
<td>-29ms</td>
</tr>
</tbody>
</table>
Bin Packing Scheduler
Architectural Implications of Function-as-a-Service Computing

Mohammad Shahrad  
Princeton University  
Princeton, USA  
mshahrad@princeton.edu

Jonathan Balkind  
Princeton University  
Princeton, USA  
jbalkind@princeton.edu

David Wentzlaff  
Princeton University  
Princeton, USA  
wentzlaf@princeton.edu

ABSTRACT

Serverless computing is a rapidly growing cloud application model, popularized by Amazon’s Lambda platform. Serverless cloud services provide fine-grained provisioning of resources, which scale automatically with user demand. Function-as-a-Service (FaaS) applications follow this serverless model, with the developer providing their application as a set of functions which are executed in response to a user- or system-generated event. Functions are designed to be short-lived and execute inside containers or virtual machines, introducing a range of system-level overheads. This paper studies the architectural implications of this emerging paradigm. Using the commercial-grade Apache OpenWhisk FaaS platform on real servers, this work investigates and identifies the architectural implications of FaaS serverless computing. The workloads, along with the way that FaaS inherently interleaves short functions from many tenants frustrates many of the locality-preserving architectural structures common in modern processors. In particular, we find that: FaaS containerization brings up to 20x slowdown compared to native execution, cold-start can be over 10x a short function’s

Figure 1: We characterize the server-level overheads of Function-as-a-Service applications, compared to native execution. This contrasts with prior work [2–5] which focused on platform-level or end-to-end issues, relying heavily on reverse engineering of commercial services’ behavior.
built for the Enterprise and Research

powers IBM Cloud Functions, Adobe I/O Runtime, Naver, Nimbellla, …
Figure courtesy of Mohammad Shahrad.
Latency Breakdown

@rabbah - Workshop on Serverless Computing 2019

Figure courtesy of Mohammad Shahrad.
Latency Breakdown

Figure courtesy of Mohammad Shahrad.
Latency Breakdown

- **Under-invoked**
  - Function initialization
  - Wait time in queue
  - Execution time

- **Balanced**
  - Function initialization
  - Wait time in queue
  - Execution time

- **Over-invoked**
  - Function initialization
  - Wait time in queue
  - Execution time

(A) Cold Starts
(B) Emptying Queue
(C) Increasing Queue
(D) Large Variations
Serverless Elasticity

resource isolation and provisioning

isolates
$5\text{ms}$

containers
$500\text{ms}$

vms
$50\text{s}$

unikernels
$100\text{ms}$

micro-vms
A RISCy Analogy

IF → ID → EX → WR
A RISC’y Analogy

IF → ID → EX → WR

Fetch function from Object Store. Cache it for repeated execution.

Decode and determine resources to allocate for function: container, memory, CPU, GPU …

Execute the function, sending it the input arguments, and capturing its result as the output.

Write the result of the execution back to the Data Store.
A RISCy Analogy

Branch Prediction : Function Prediction
Speculation : Pre-Warming
Register Bypass : Function to Function
Serverless Contract

functions run in finite time and space

...and have transient residency
death is certain
but revival is fast
Can compositions of serverless functions be serverless functions?
The Computing Stack

- Applications
- Libraries, DSLs
- Compilers
- Runtime & OS
- ISA
- Micro Architecture
Function Composition

function A \rightarrow function B \rightarrow function C \rightarrow function D
Function Orchestration

where is “main”?

client-based scheduler

scheduler as a function

fusing scheduler function

continuation scheduling

@rabbah - Workshop on Serverless Computing 2019
Client-side Composition?

scheduler as client

explicit activations

function A  function B  function C  function D

composition cannot be further composed: substitution
Reflective Composition?

scheduler waits for functions to complete: double billing
Composition by Fusion?

monoglot and requires access to source: black box

```plaintext
let fused = [
    args => ...
    args => ...
    args => ...
]

let scheduler = functions => args =>
    functions.reduce(Function.apply, args)

let main = scheduler(fused)
```
Continuations?

the right direction, but **breaks** substitution, double billing, or black box
Serverless Trilemma

black box — double billing

let me compose services or code

charge me for functions, not scheduling

substitution

permit blocking invokes and hierarchical composition

without intrinsic support, compositions-as-functions violate at least one constraint
programming model for Serverless Composition
Composition with Combinators

\[
\text{composer.sequence(}
    \text{'}A\text{'},
    \text{'}B\text{'},
    \text{'}C\text{'},
    \text{'}D\text{'}
\text{)}
\]

github.com/apache/openwhisk-composer
Function Orchestration

```python
composer.sequence('A', 'B', 'C', 'D')
```
Control and Data Flow Combinators

|        | ![Sequence Diagram](image)
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence</strong></td>
<td><img src="image" alt="Sequence Diagram" /></td>
</tr>
<tr>
<td><strong>Branch</strong></td>
<td><img src="image" alt="Branch Diagram" /></td>
</tr>
<tr>
<td><strong>Loop</strong></td>
<td><img src="image" alt="Loop Diagram" /></td>
</tr>
<tr>
<td><strong>Parallel</strong></td>
<td><img src="image" alt="Parallel Diagram" /></td>
</tr>
</tbody>
</table>

[github.com/apache/openwhisk-composer](https://github.com/apache/openwhisk-composer)
From Functions to Serverless Applications

```javascript
composer.sequence(
  `\!/whisk.system/utils/echo`,
  `\!${prefix}/extract`,
  `\!${prefix}/fetch.job.id`,
  composer.retain(
    composer.sequence(
      composer.retry(3, `\!${prefix}/fetch.log.url`),
      `\!${prefix}/analyze.log`),
      ({
        result,
        params
      }) => Object.assign(result, params),
    `\!${prefix}/format.for.slack`,
    composer.retain(
      composer.value(slackConfig)),
      ({
        result,
        params
      }) => Object.assign(result, params),
    `\!/whisk.system/slack/post`))
```

github.com/rabbah/travis-to-slack
The Computing Stack and Serverless Abstraction Gaps

Applications
Libraries, DSLs
Compilers
Runtime & OS
ISA
Micro Architecture
The Computing Stack and Serverless Abstraction Gaps

Applications
Libraries, DSLs
Compilers
Runtime & OS
ISA
Cloud Providers as Commodity
Serverless Functions
A computer is not just a CPU
The Serverless Instruction Set

serverless compute

serverless memory
The Serverless Instruction Set

- Low Latency
- Function Memory
- Function-Function Networking
- Cloud Providers as Commodity
The Serverless Instruction Set

*the dawn of the Cloud Computer*

- Low Latency
- Function Memory
- Function-Function Networking
- Cloud Providers as Commodity
1964: Invention of the Instruction Set Architecture
Serverless & Stateful
Read State / Memory

Logic

Write State / Memory
This is hard.

functions have **transient residency**

**transient** residency → **no data locality**
Session: Formalization Chair(s): Eric Koskinen

Unfortunately, the serverless computing abstraction exposes several low-level operational details that make it hard for programmers to write and reason about their code. This paper sheds light on this problem by presenting $\lambda_\lambda$, an operational semantics of the essence of serverless computing. Despite being a small (half a page) core calculus, $\lambda_\lambda$ models all the low-level details that serverless functions can observe. To show that $\lambda_\lambda$ is useful, we present three applications. First, to ease reasoning about code, we present a simplified naive semantics of serverless execution and precisely characterize when the naive semantics and $\lambda_\lambda$ coincide. Second, we augment $\lambda_\lambda$ with a key-value store to allow reasoning about stateful serverless functions. Third, since a handful of serverless platforms support serverless function composition, we show how to extend $\lambda_\lambda$ with a composition language and show that our implementation can outperform prior work.

Link to Publication: https://people.cs.umass.edu/~brun/pubs/pubs/Jangda19oopsla.pdf

Abhinav Jangda  
University of Massachusetts Amherst

Donald Pinckney  
University of Massachusetts Amherst

Yuriy Brun  
University of Massachusetts Amherst

Arjun Guha  
University of Massachusetts, Amherst

United States
Serverless Operational Semantics

Serverless Functions \((f, \Sigma, \text{recv}, \text{step}_f, \text{init})\)

Functions \(F := \cdots\)

Function name \(f \in F\)

Internal states \(\Sigma := \cdots\)

Initial state \(\text{init} \in F \rightarrow \Sigma\)

Receive event \(\text{recv}_f \in \nu \times \Sigma \rightarrow \Sigma\)

Internal step \(\text{step}_f \in F \times \Sigma \rightarrow \Sigma \times t\)

Values \(\nu := \cdots\)

Values \(\nu := \cdots\)

Commands \(t := \epsilon\)

\[.\]

Serverless Platform

Request ID \(x := \cdots\)

Instance ID \(y := \cdots\)

Execution mode \(m := \text{idle} \quad \text{Idle} \quad \text{BEGIN}\)

| \(x\) | Process

Transition labels \(\ell := \text{Internal}\)

| \(x\) | \(\text{receive}(x)\) \(\text{Receive}\) \(\text{END}\)

| \(x\) | \(\text{stop}(x, \nu)\) \(\text{Respond}\) \(\text{DROP}\)

Components \(C := \{f, m, \sigma, y\}\)

Component set \(C := \{C_1, \cdots, C_n\}\)
“For its entire history, distributed computing research modeled capacity as fixed but time as unlimited.

With serverless time is limited, but capacity is effectively infinite.

This only changes everything.”

Dr. Tim Wagner
Amazon Lambda “inventor”