The Dawn of the Cloud Computer

Rodric @Rabbah

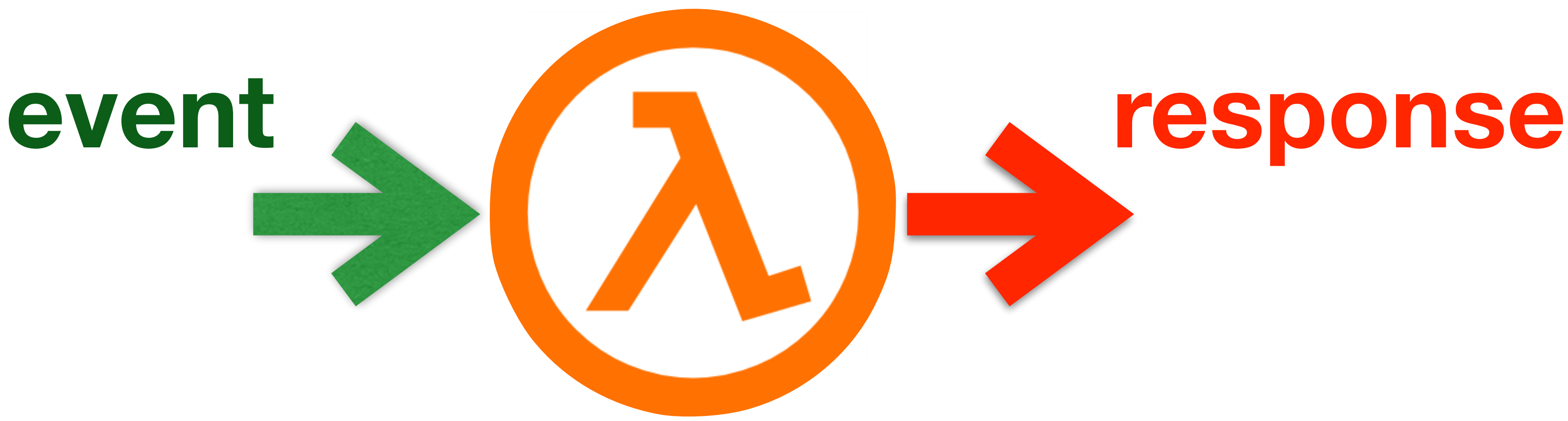
nimbella.com

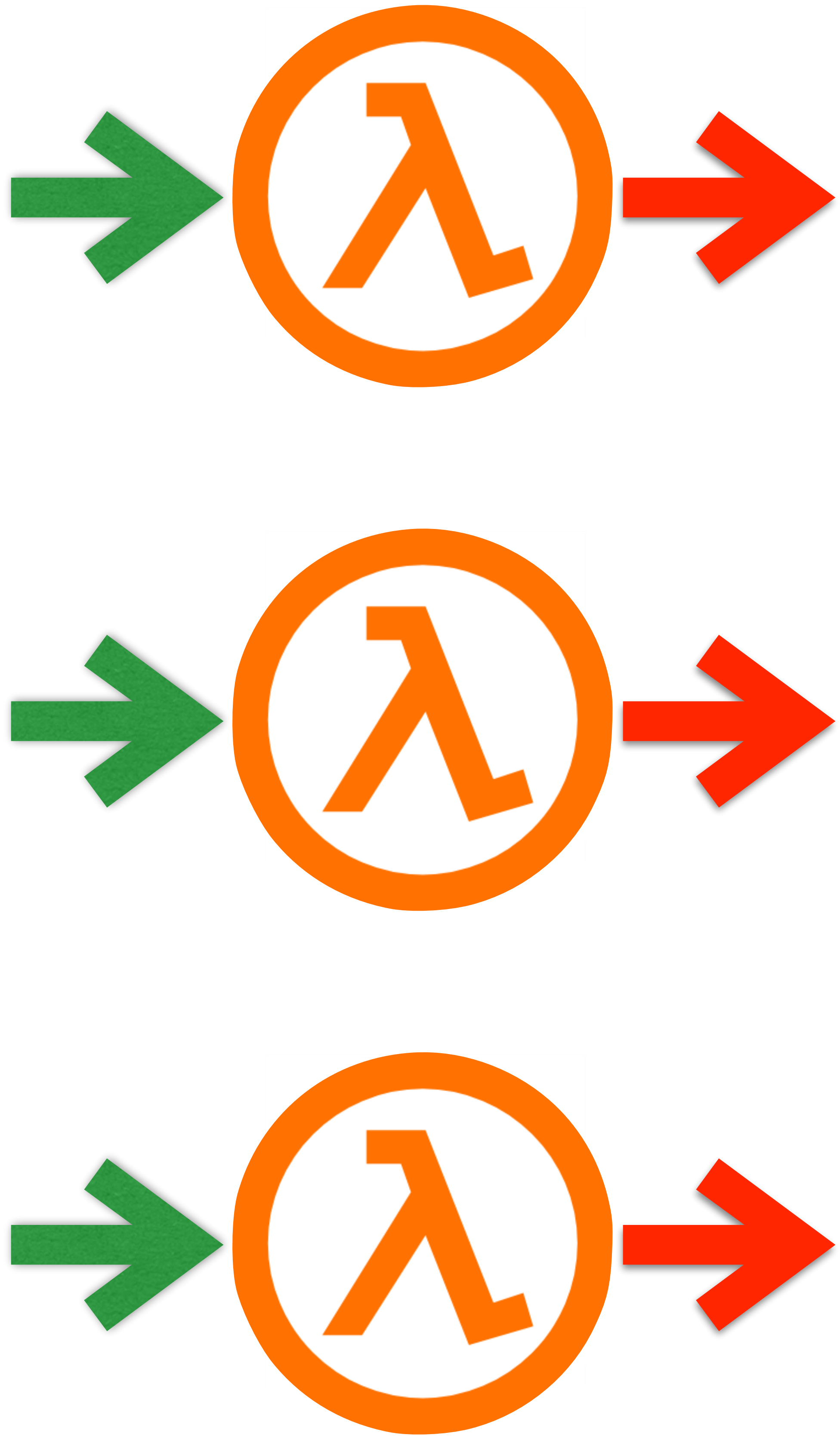
Fifth International Workshop on Serverless Computing
WoSC 2019

5 years ago Amazon announced…

instantly reactive **functions**

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


event
response



“example”

```
> let hello = …
> open bit.ly/hello-fn
```

no **Server** logic

```
server.route('/hello',
  let main = () => ({
    msg: "Hello World"
  })
)
server.listen(port)
```

no **Server** at all

```
server.route('/hello',
    let main = () => ({
      msg: "Hello World"
    })
)
server.listen(port)
```

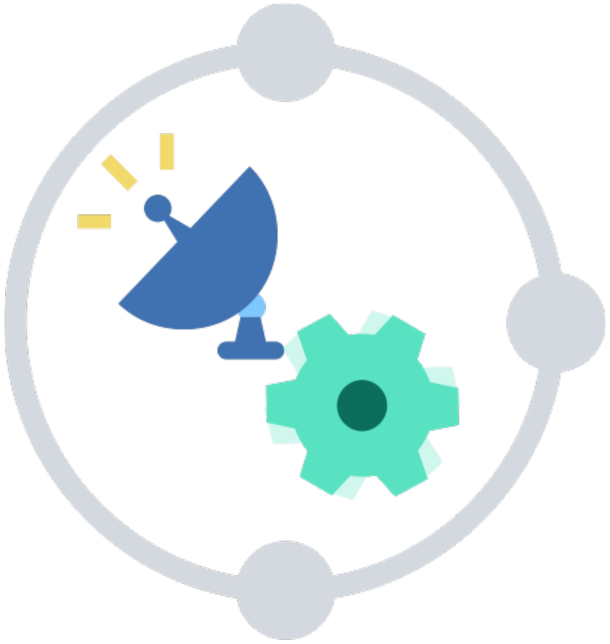
highly **concurrent** by default

```
Running 10s test @ https://apigcp.nimbella.io/api/v1/web/rabbahgm-rg0c4xagzcl/default/hello.json
  10 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    43.86ms   30.24ms 284.36ms   91.30%
    Req/Sec    25.36      7.86    40.00     83.12%
  2482 requests in 10.06s, 1.19MB read
Requests/sec:    246.64
Transfer/sec:    120.90KB
```


node
JS
Swift
GO
python
docker
Java
Ruby
php

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 WINSTEIN

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads

Sadjad Fouladi, Riad S. Wahby, and Brennan Shacklett, *Stanford University;* Karthikeyan Vasuki Balasubramaniam, *University of California, San Diego;* William Zeng, *Stanford University;* Rahul Bhalerao, *University of California, San Diego;* Anirudh Sivaraman, *Massachusetts Institute of Technology;* George Porter, *University of California, San Diego;* Keith Winstein, *Stanford University*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Occupy 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

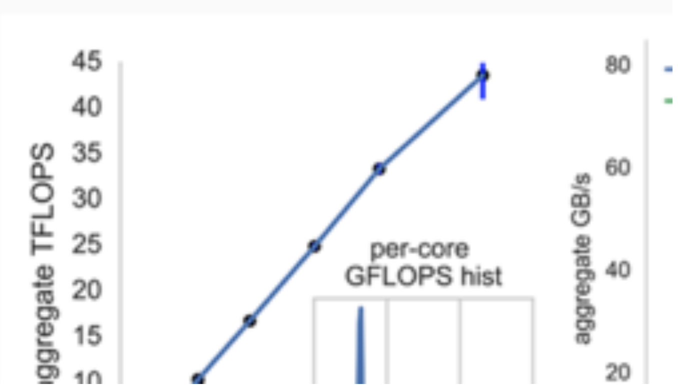
ABSTRACT

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 ahead of time decide on instance type, cluster size, pricing model, programming model, and task granularity.

PyWren: Real-time Elastic Execution

PyWren is a system we built to enable incredibly scalable executio on the cloud using AWS Lambda (and other "serverless" framewor mean it -- you can nearly-instantly run your code on literally thous overhead, all billed in 100ms-increments.

PyWren began as a series of exploratory blog posts looking at the compute scaling and IO scaling of Amazon's cloud services, and blossomed into a joint project between



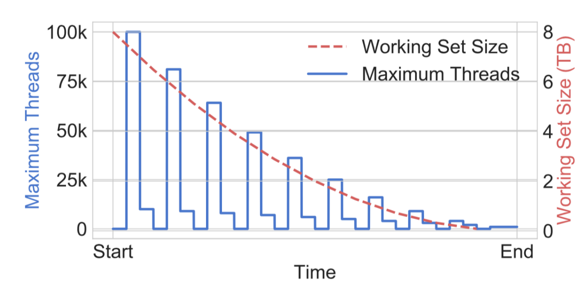
numpywren: Serverless Linear Algebra

Vaishaal Shankar[1], Karl Krauth[1], Qifan Pu[1], Eric Jonas[1], Shivaram Venkataraman[2], Ion Stoica[1], Benjamin Recht[1], and Jonathan Ragan-Kelley[1]

[1]UC Berkeley
[2]UW 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



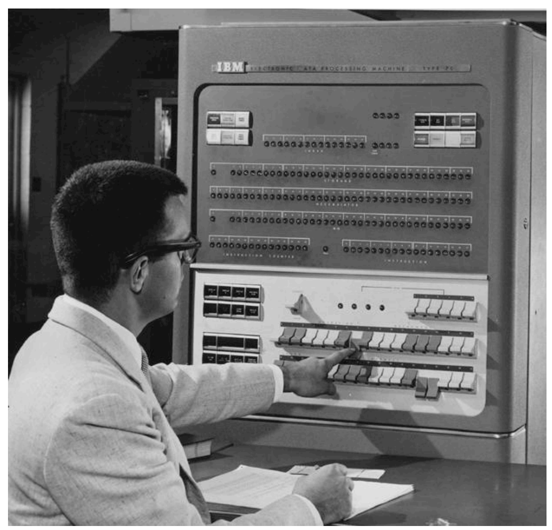
It is not all **academic.**

$100M+ **investor** backed **serverless** startups in 2018

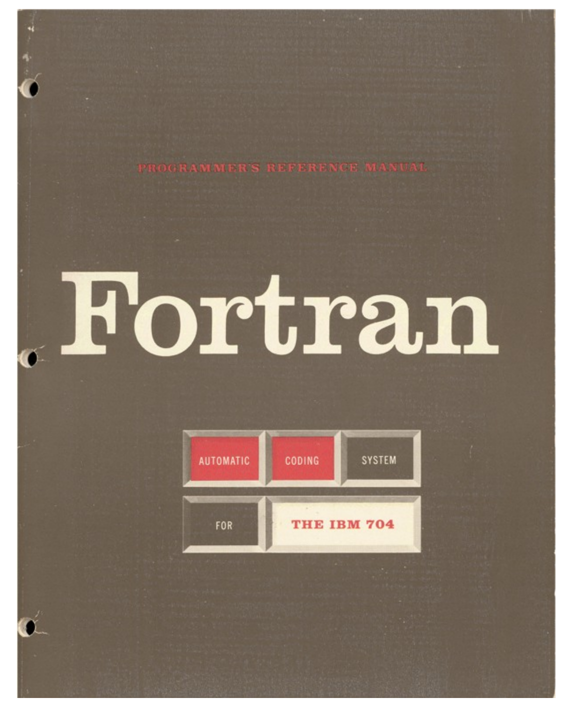
The cloud computing landscape is changing.

The cloud computing landscape ~~is changing.~~ has changed.

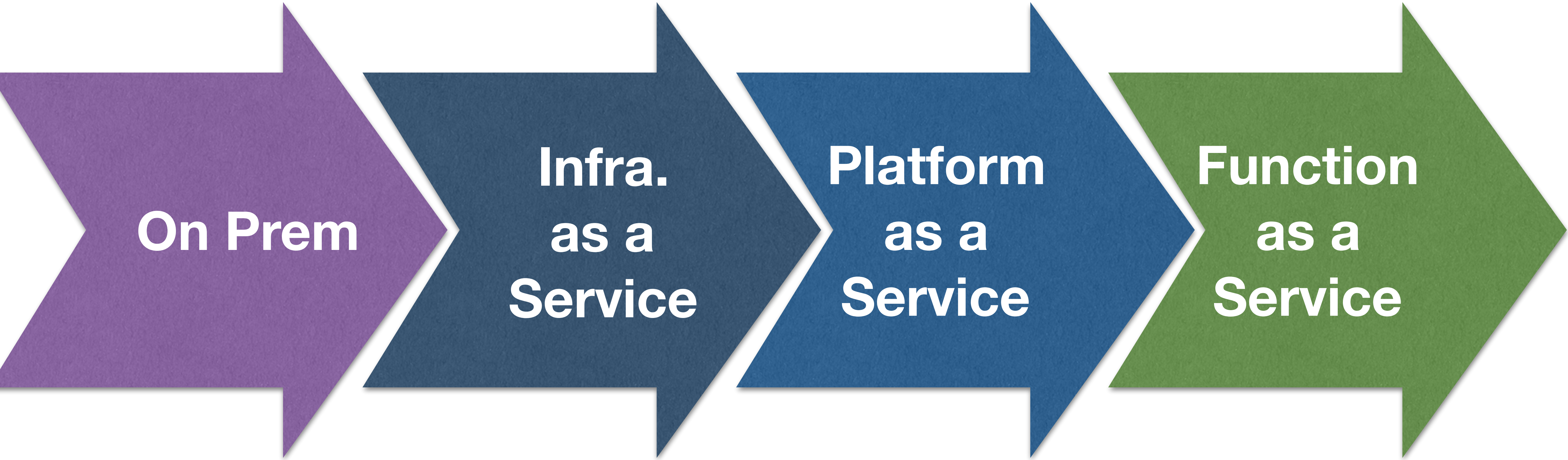
Increasing **automation** & **abstraction** are engrained in the **history** of computing.



IBM 704
1954



Increasing **automation** & **abstraction** are engrained in the **history** of computing.



$ / machine

$ / hour

$ / request

2015

2 **Billion** Lambdas / day

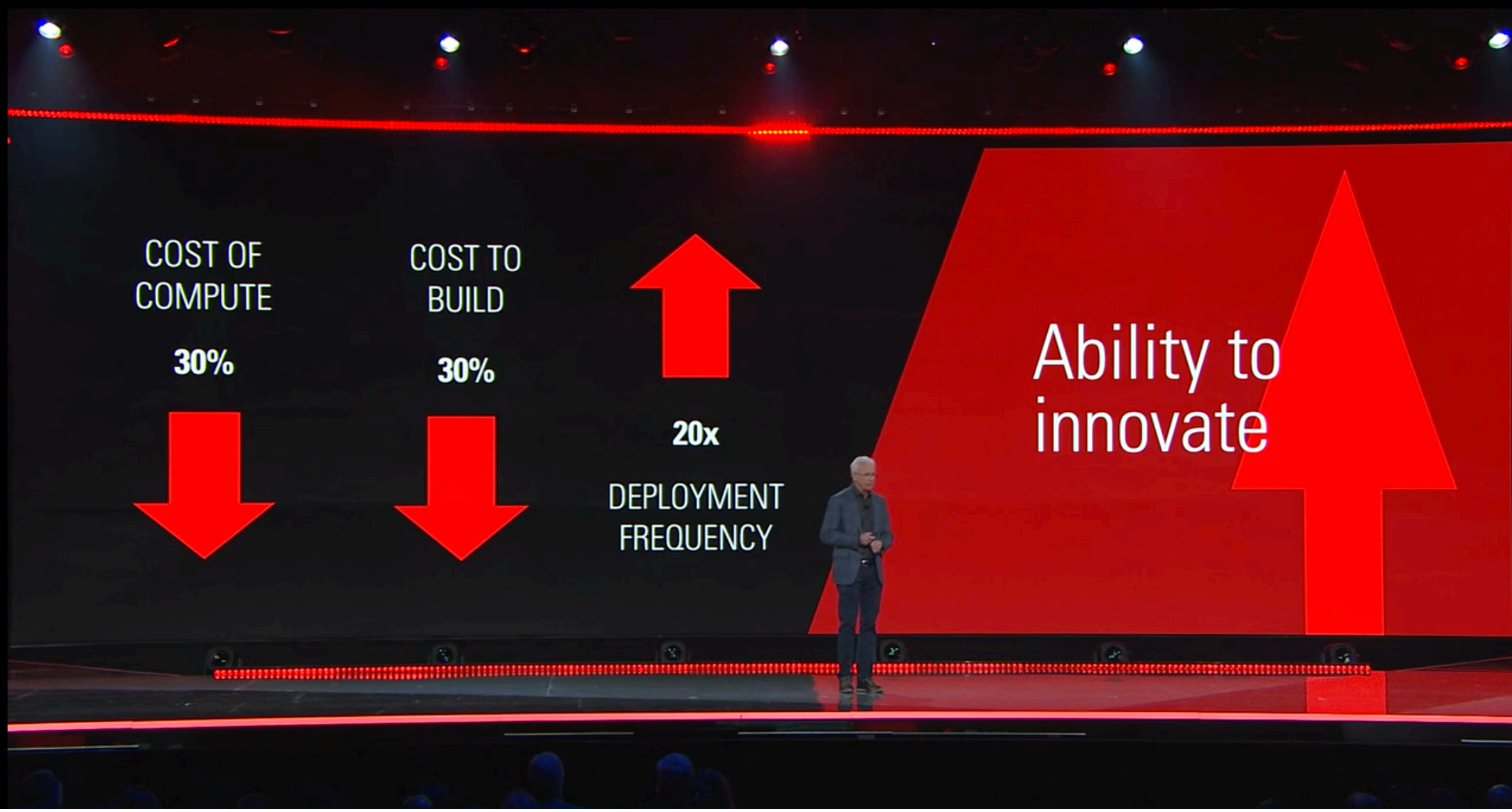
2019

2+ **Trillion** / month

Serverless is **inevitable**.

AWS re:Invent 2019

AWS re:Invent Livestream **Dr. Werner Vogels**

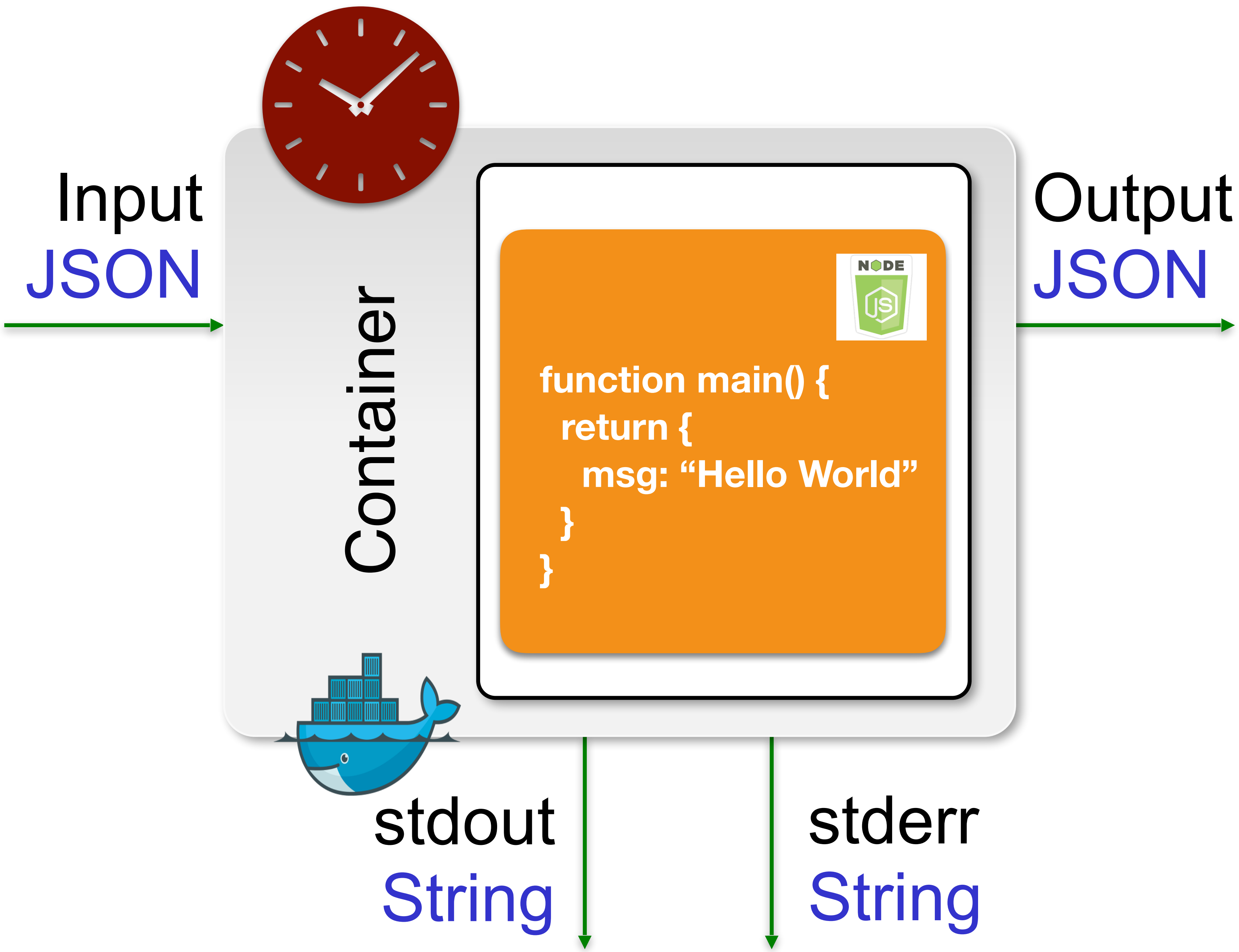


My Serverless Conjecture

The number of ***servers***
managed *by an organization*
will decrease in ***half every 2 years****.*

there are limits…
of course

Function Isolation

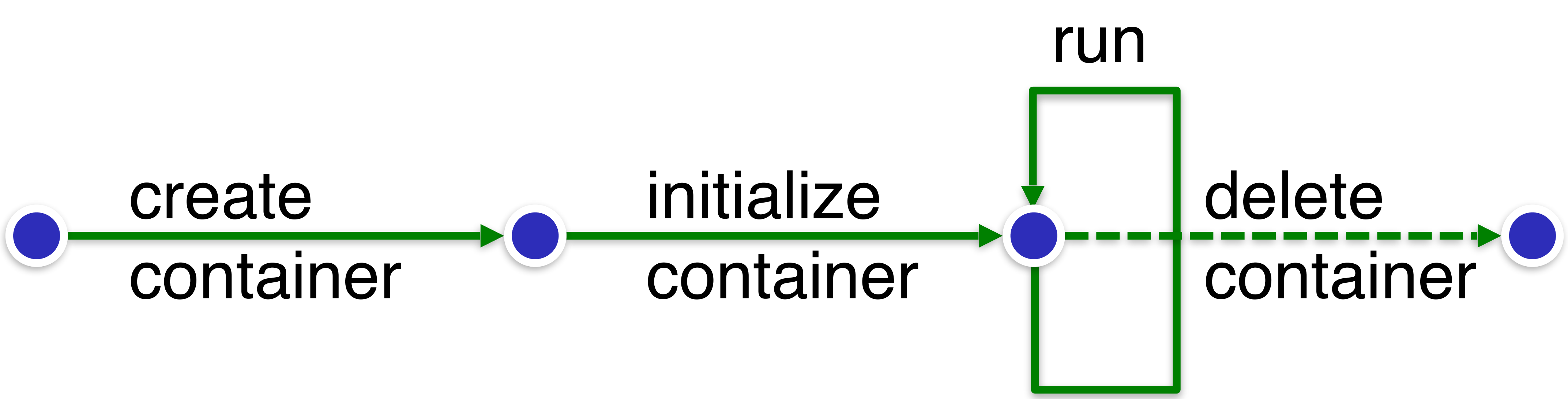


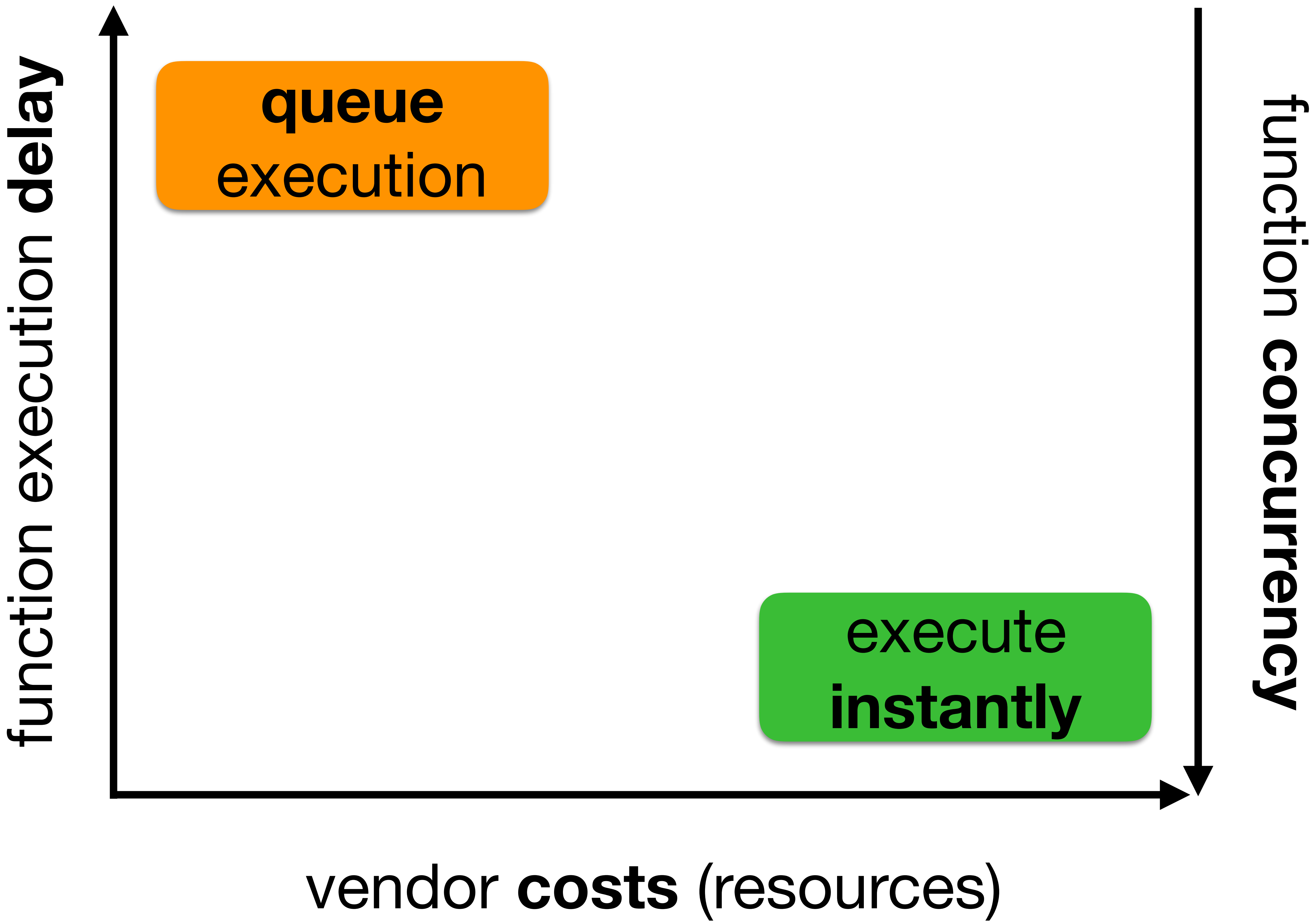
Serverless Elasticity

resource isolation and provisioning

containers
500ms

Container Lifecycle




function execution delay
queue execution
execute instantly
function concurrency
vendor costs (resources)

Serverless Tensions

scale infinitely execute instantly	**vs.**	control costs 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**: queuing latency ≈ 0

The system is over-provisioned.

Serverless Contract

A ≈ **D**: queuing latency ≈ 0

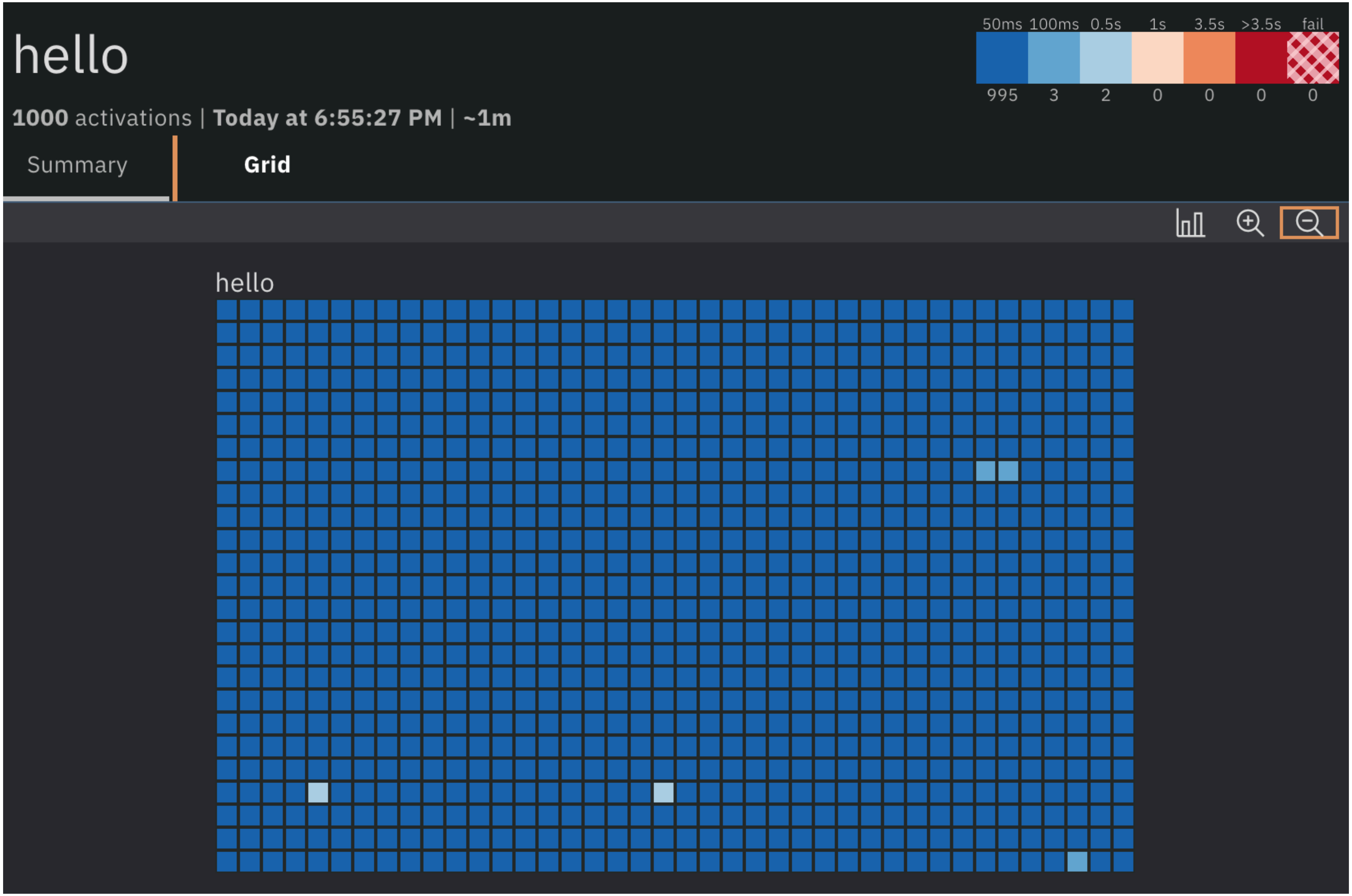
Balanced but difficult to achieve with dynamic load.

Serverless Contract

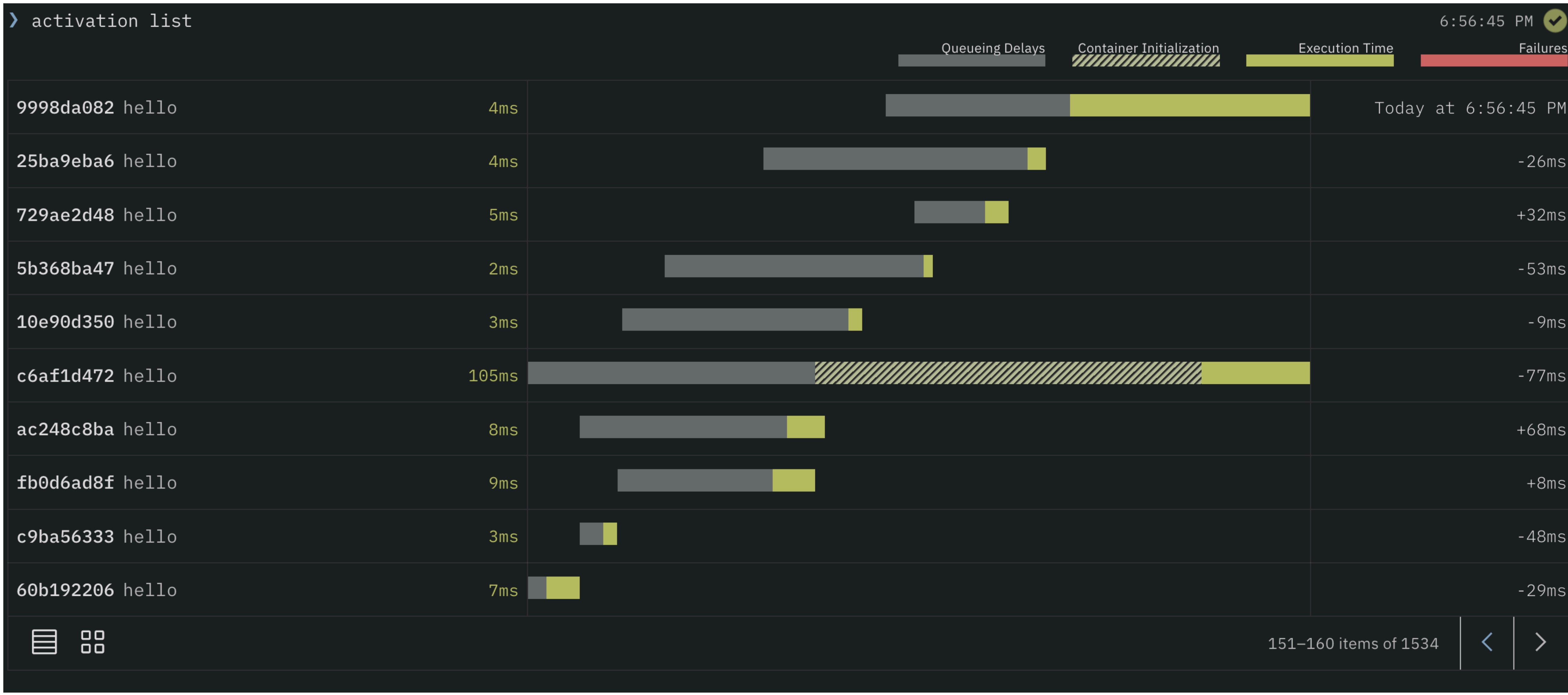
A $>$ **D**: queuing latency $\propto$ mismatch

The system is under-provisioned.

Cold Starts

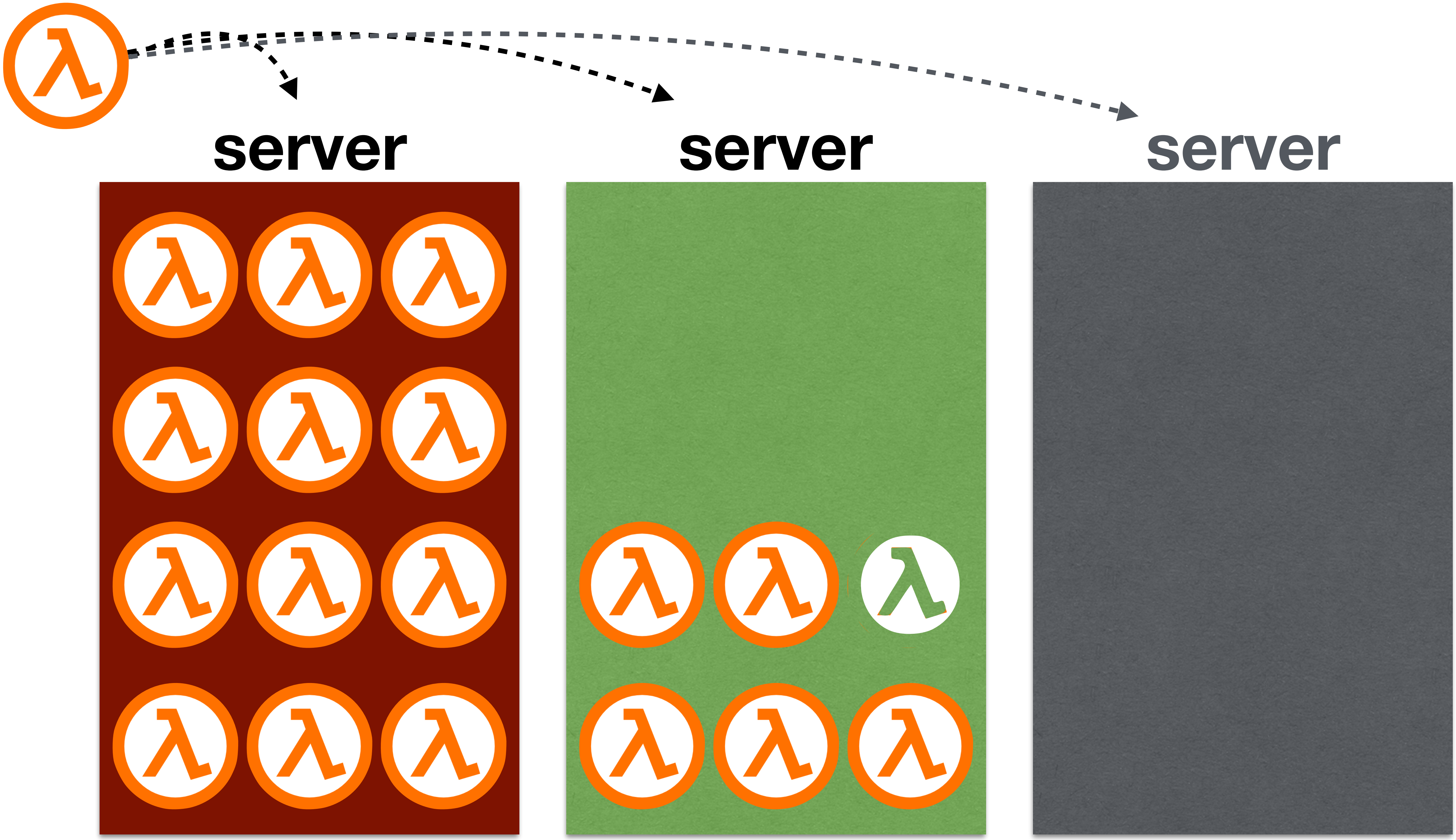

hello
50ms 100ms 0.5s 1s 3.5s >3.5s fail
995 3 2 0 0 0 0
1000 activations | Today at 6:55:27 PM | ~1m
Summary
Grid
hello

Wait Time



❯ activation list
6:56:45 PM
Queueing Delays
Container Initialization
Execution Time
Failures
9998da082 hello 4ms Today at 6:56:45 PM
25ba9eba6 hello 4ms -26ms
729ae2d48 hello 5ms +32ms
5b368ba47 hello 2ms -53ms
10e90d350 hello 3ms -9ms
c6af1d472 hello 105ms -77ms
ac248c8ba hello 8ms +68ms
fb0d6ad8f hello 9ms +8ms
c9ba56333 hello 3ms -48ms
60b192206 hello 7ms -29ms
151–160 items of 1534

Bin Packing Scheduler



MICRO 2019

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

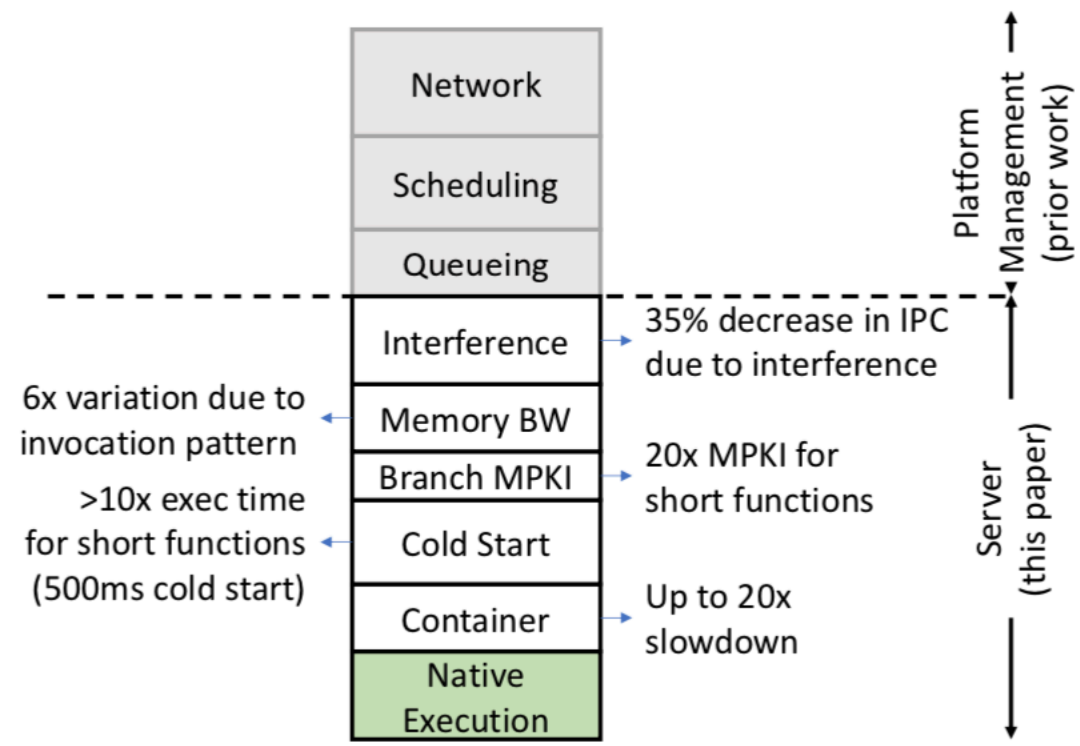


Figure 1: We characterize the server-level overheads of Function-as-a-Service applications, compared to native execution. This contrasts with prior work [2–5] which focused on platform-level or end-to-end issues, relying heavily on reverse engineering of commercial services' behavior.

APACHE
OpenWhisk

github.com/apache/openwhisk

APACHE OpenWhisk

4.4K+ stars
845+ forks
165+ contributors

built for the Enterprise and Research

powers IBM Cloud Functions,
Adobe I/O Runtime, Naver, Nimbella, …

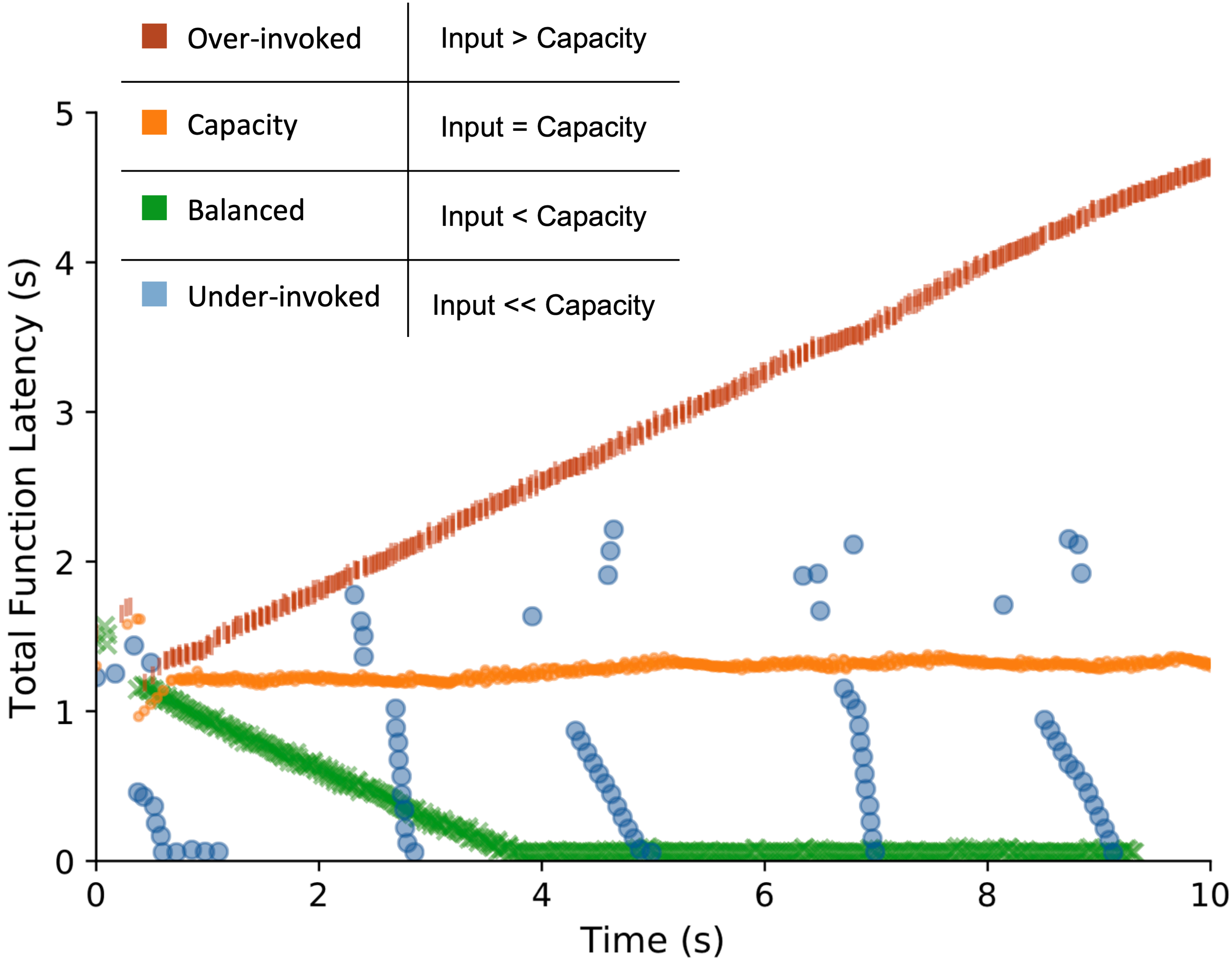


Figure courtesy of Mohammad Shahrad.

Latency Breakdown

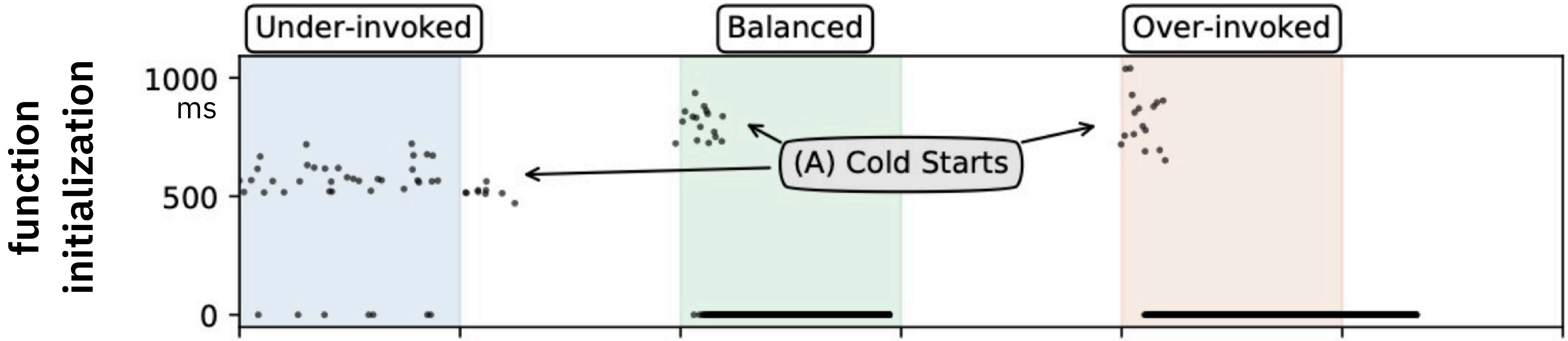


Figure courtesy of Mohammad Shahrad.

Latency Breakdown

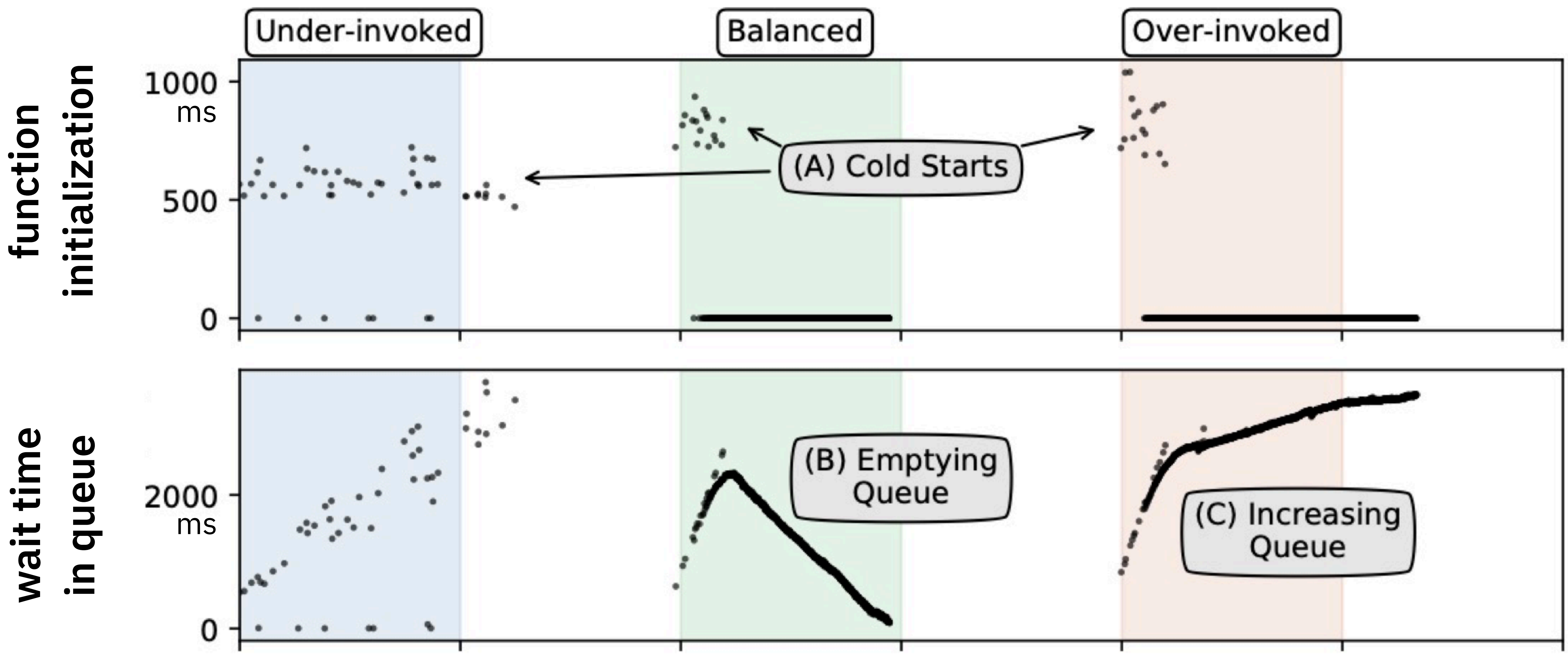
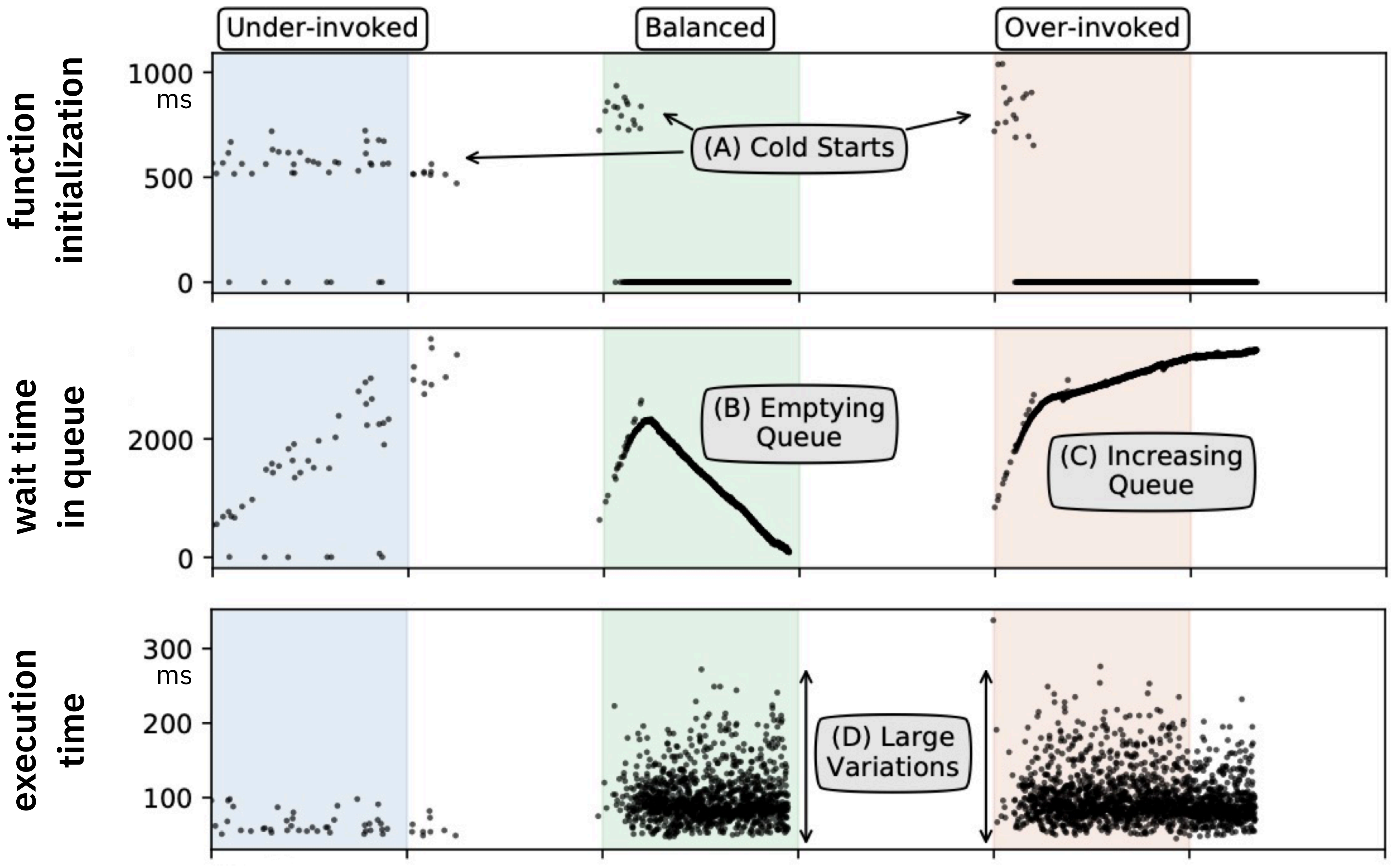


Figure courtesy of Mohammad Shahrad.

Latency Breakdown



Serverless Elasticity

resource isolation and provisioning

isolates
5ms

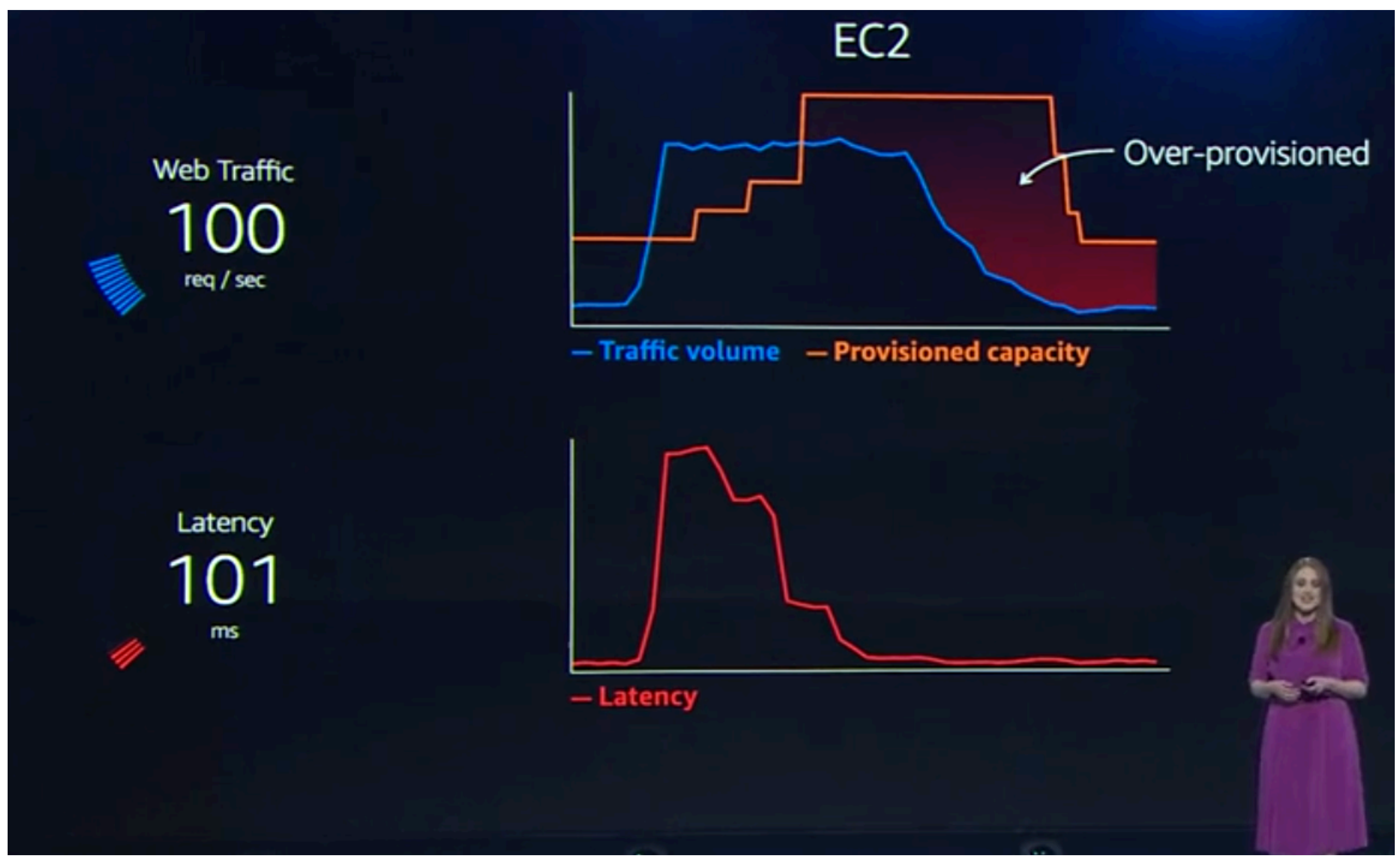
containers
500ms

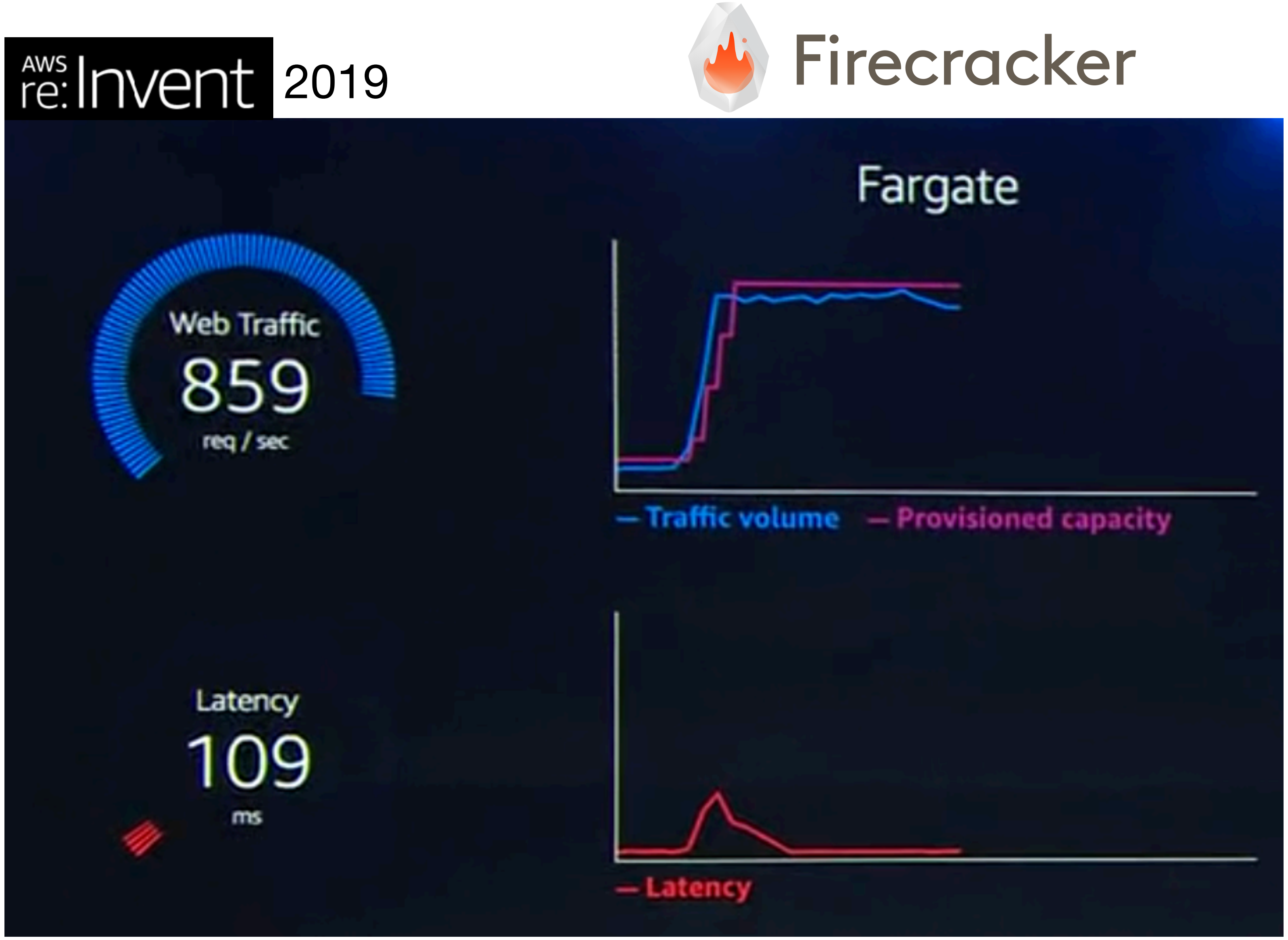
vms
50s

unikernels **micro-vms**

100ms

AWS re:Invent 2019




AWS re:Invent 2019
Firecracker
Fargate
Web Traffic
859
req / sec
— Traffic volume
— Provisioned capacity
Latency
109
ms
— Latency

A **RISC***y* 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 **RISC***y* 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

Onward! '17

Can **compositions** *of serverless functions* ***be serverless functions?***

The Computing Stack

Applications

Libraries, DSLs

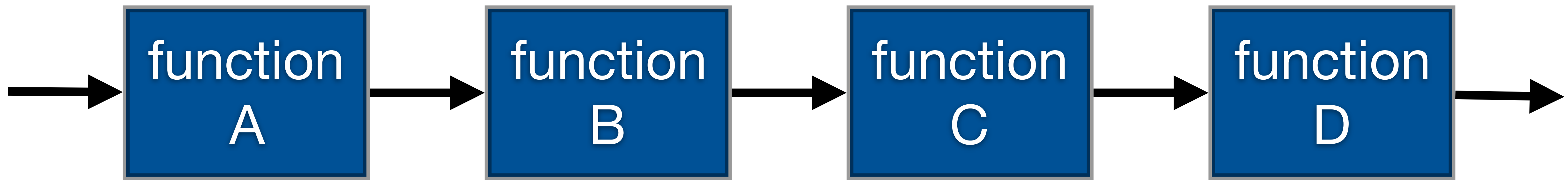
Compilers

Runtime & OS

ISA

Micro Architecture

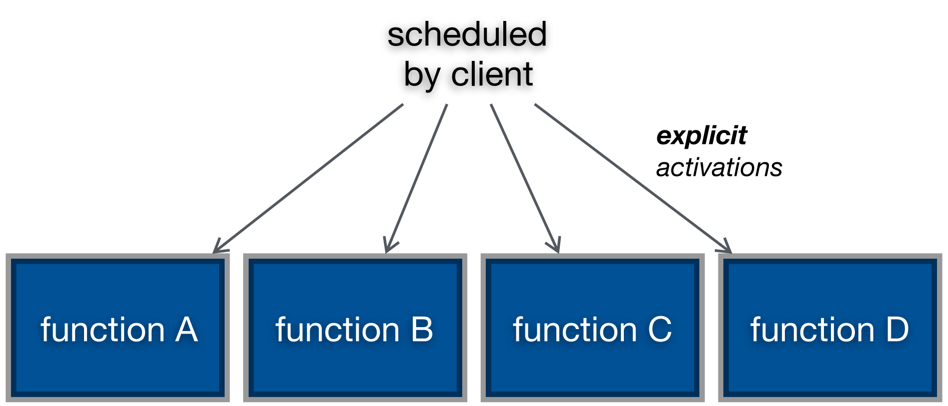
Function Composition



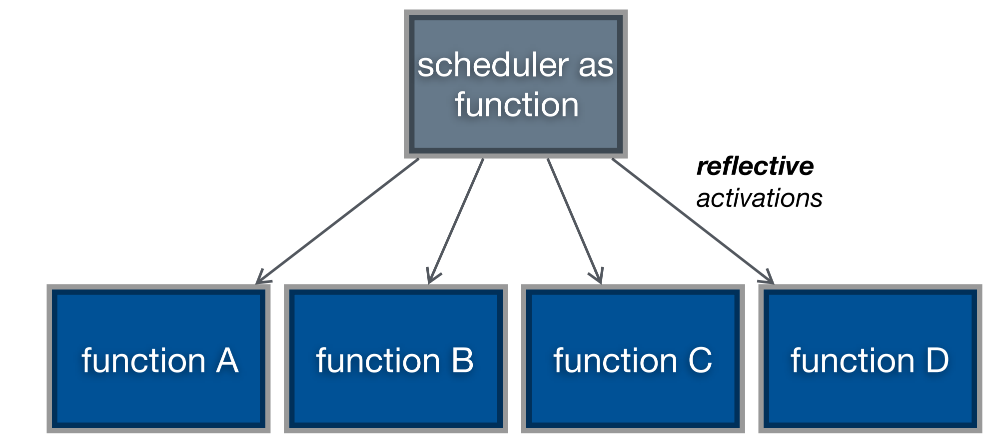
Function Orchestration

where is “**main**”?

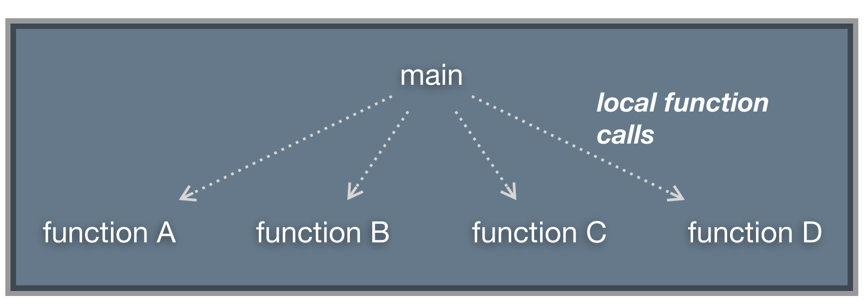
client-based scheduler



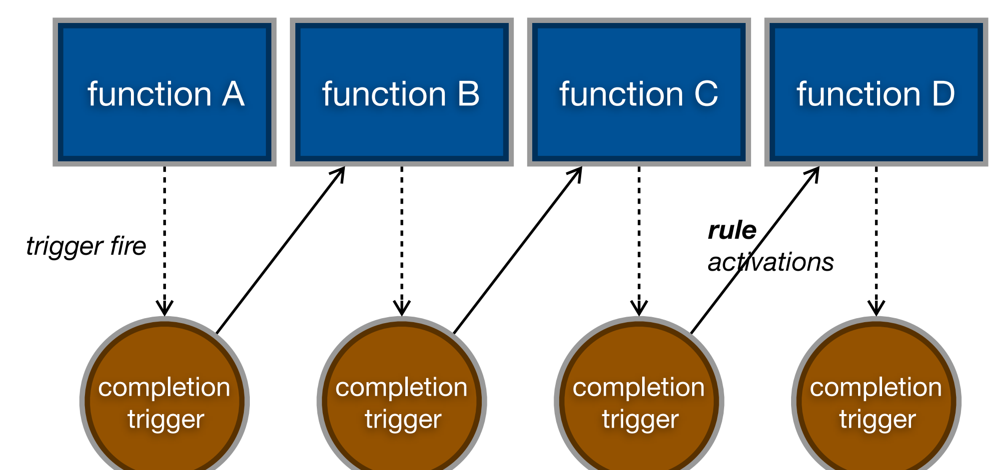
scheduler as a function



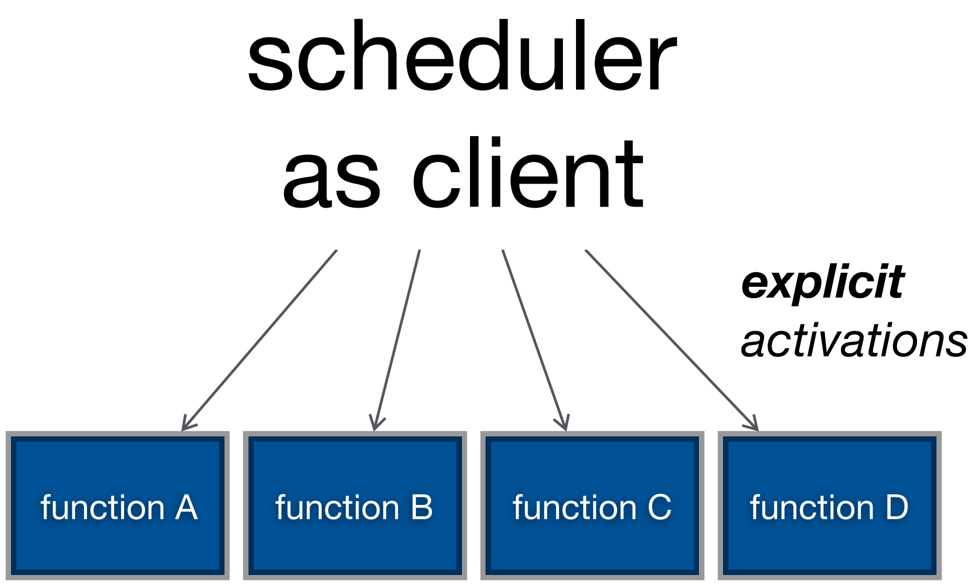
fusing scheduler function



continuation scheduling

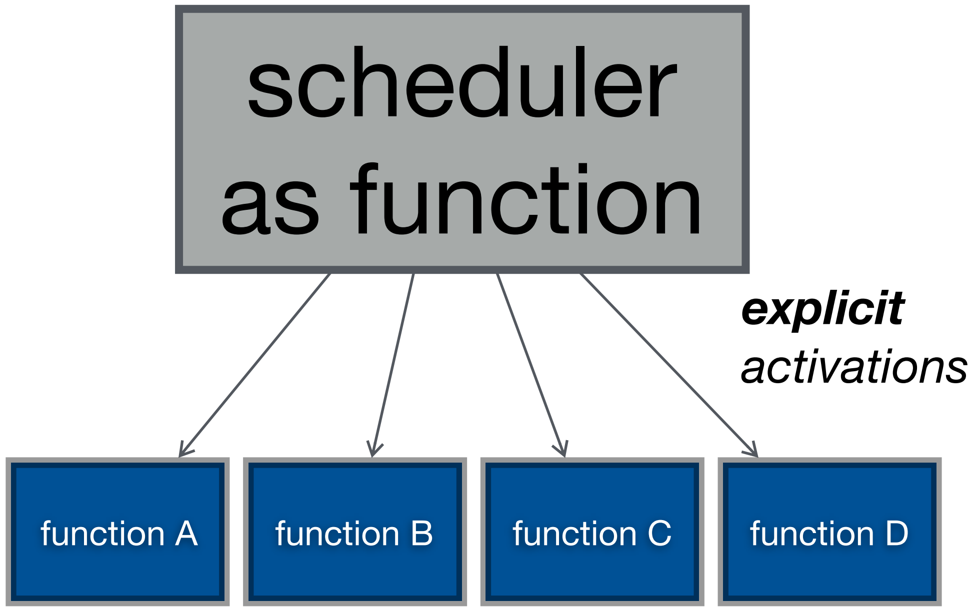


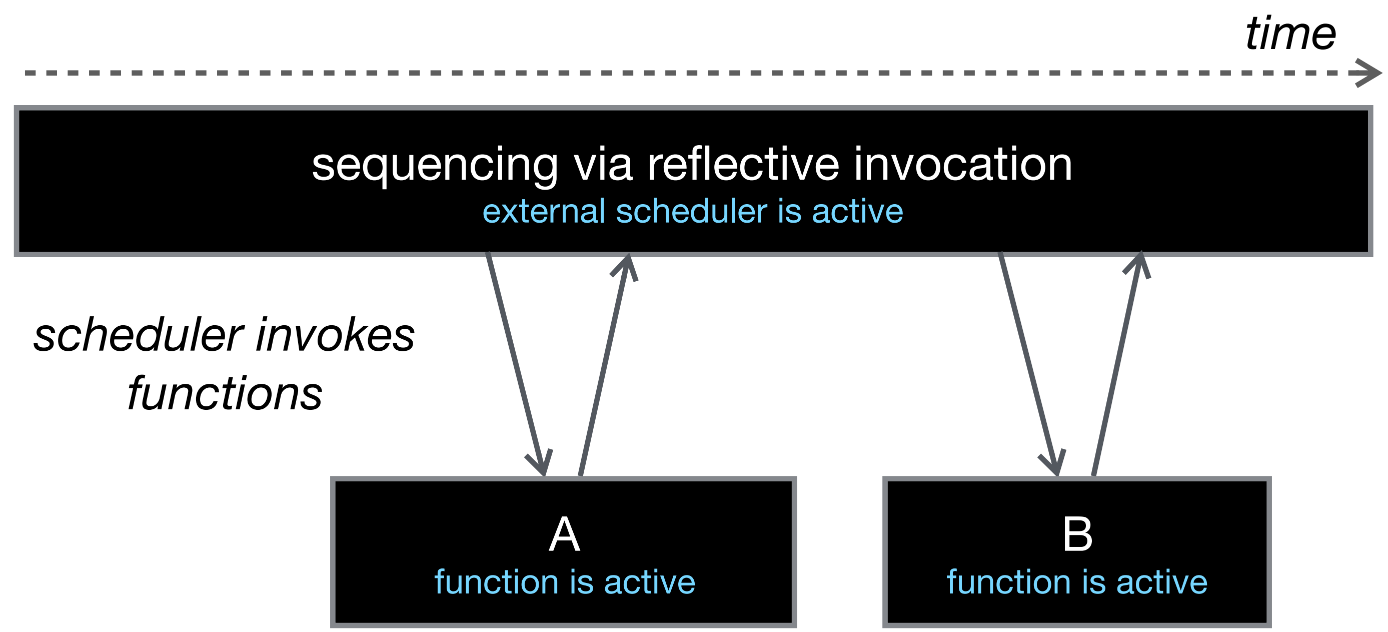
Client-side Composition?



composition cannot be further composed:
substitution

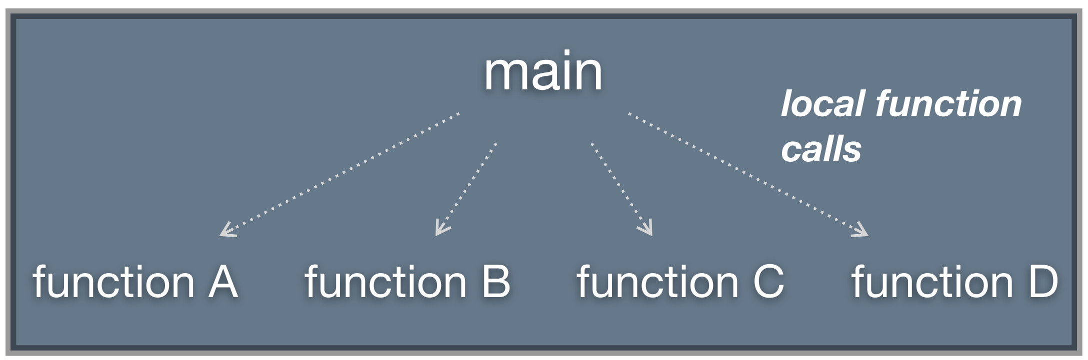
Reflective Composition?





scheduler waits for functions to complete:
double billing

Composition by Fusion?

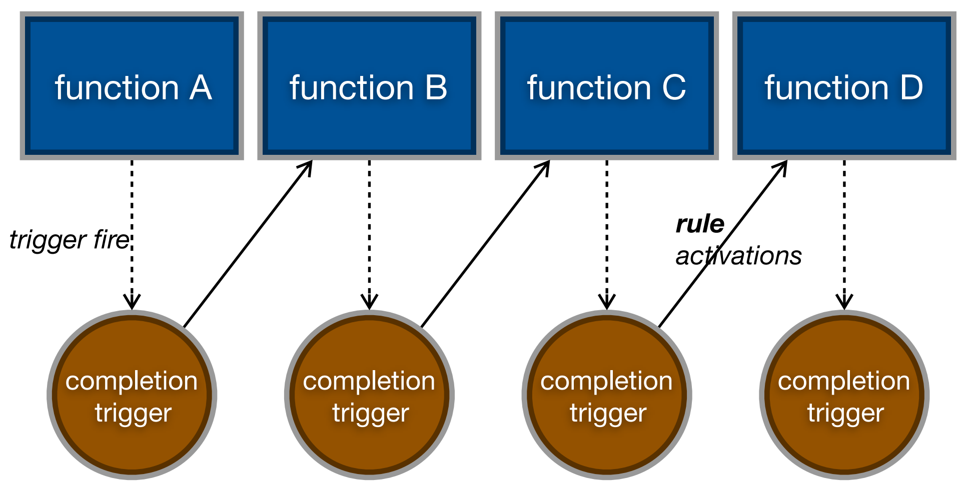


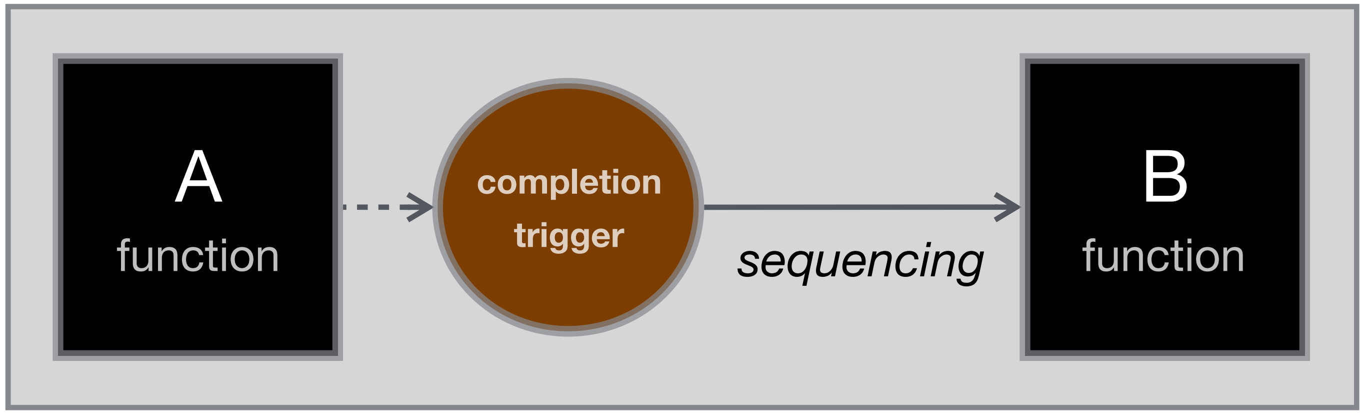

```
let fused = [
   args => ...,        inlined code for
   args => ...,        sequence
   args => ...
]


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

let main = scheduler(fused)
```

monoglot and requires access to source:

black box

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**

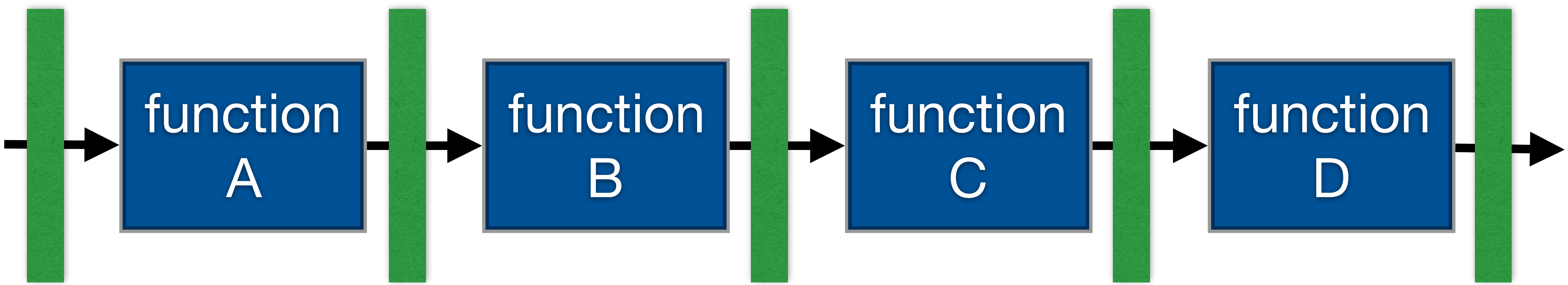
Composition with **Combinators**

```
composer.sequence(
   ‘A’,
   ‘B’,
   ‘C’,
   ‘D’
)
```

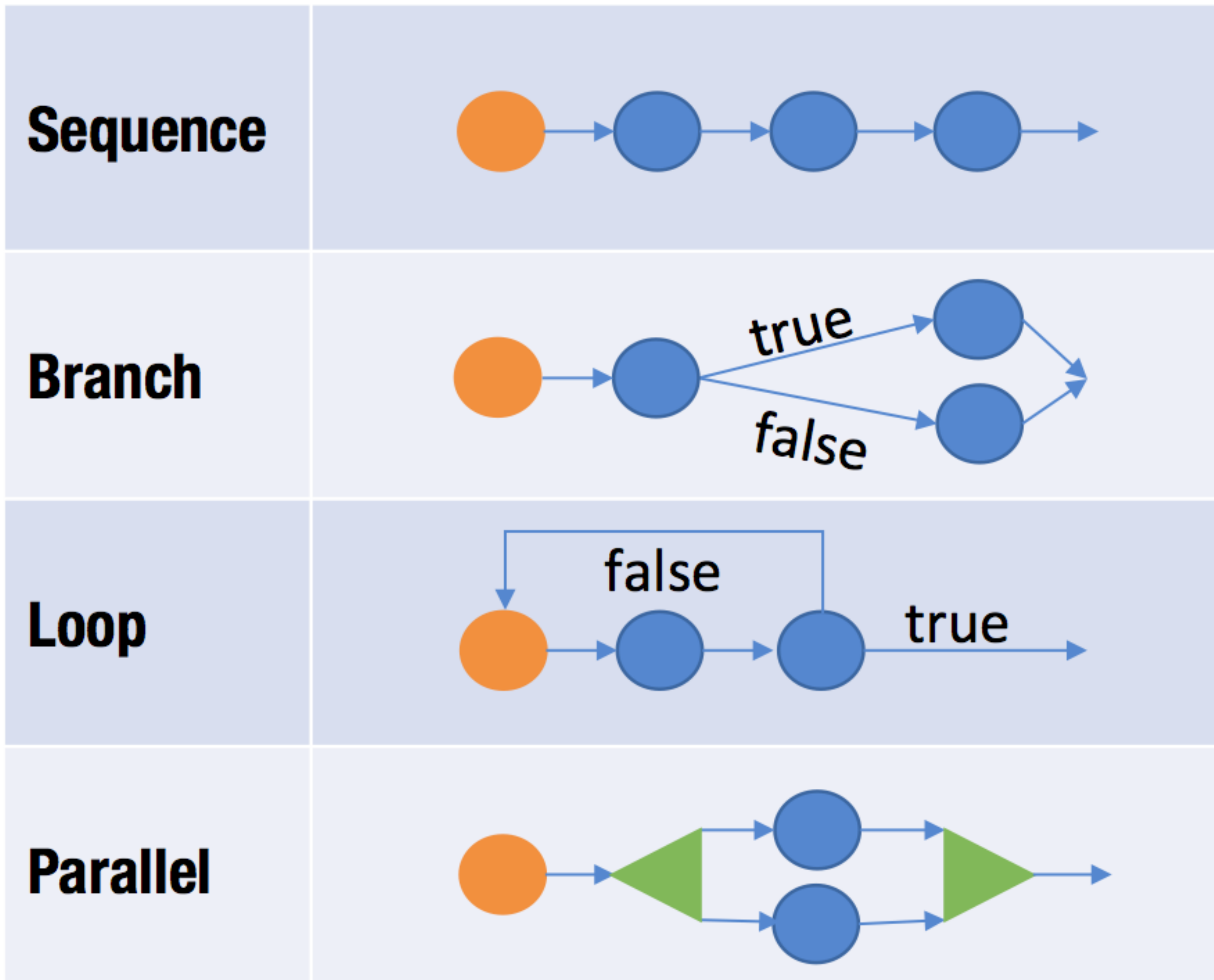
APACHE OpenWhisk *github.com/apache/openwhisk-composer*

Function Orchestration

```
composer.sequence('A', 'B', 'C', 'D')
```



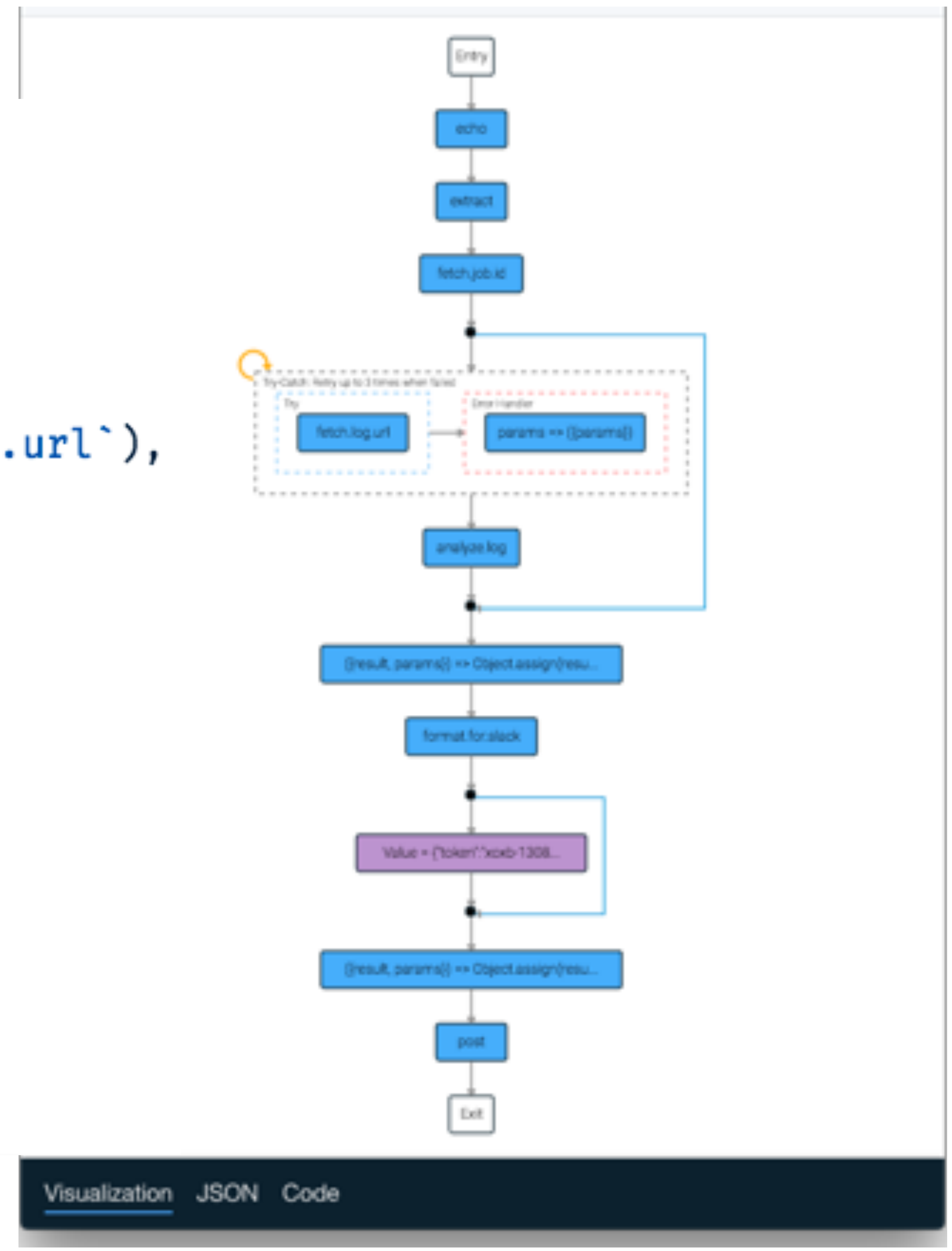
Control and Data Flow Combinators



github.com/apache/openwhisk-composer

From Functions to Serverless Applications

```
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

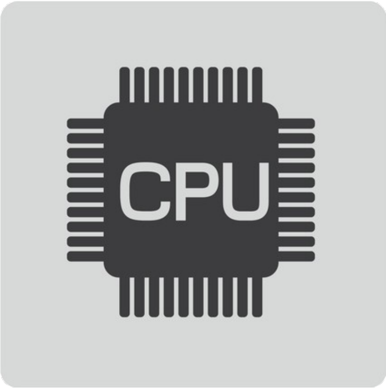
The Computing Stack and Serverless Abstraction Gaps

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

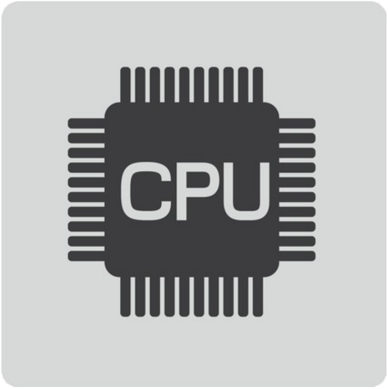
The Computing Stack and Serverless Abstraction Gaps

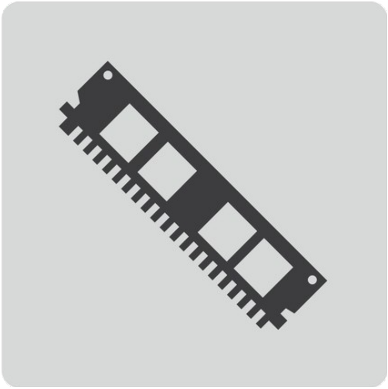
- Applications
- Libraries, DSLs
- Compilers
- Runtime & OS
- ISA
- Cloud Providers as Commodity

Serverless Functions

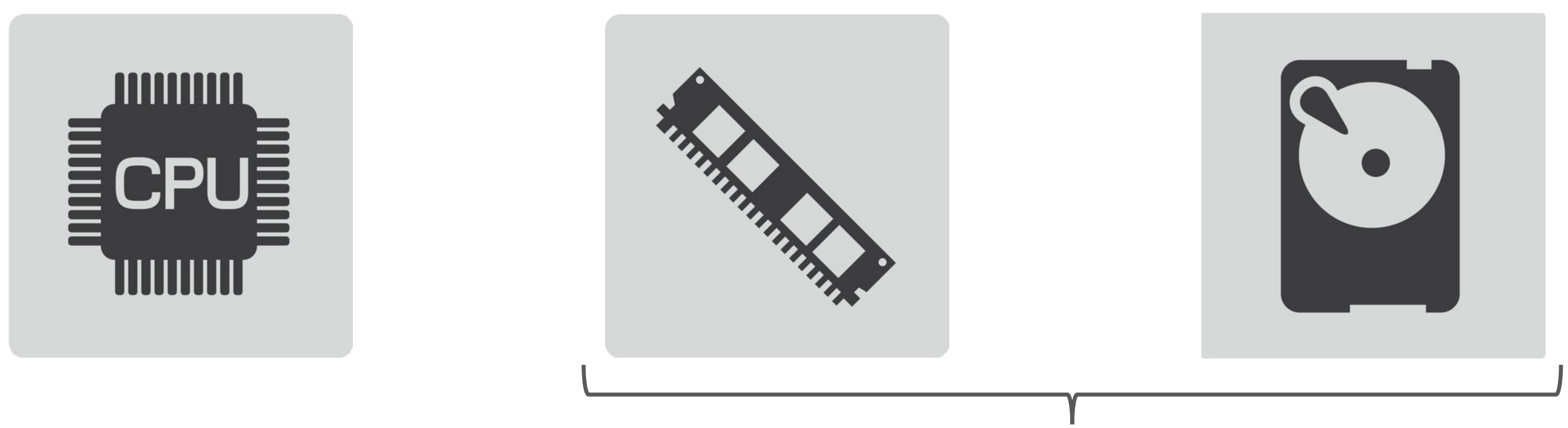


A computer is not just a CPU





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

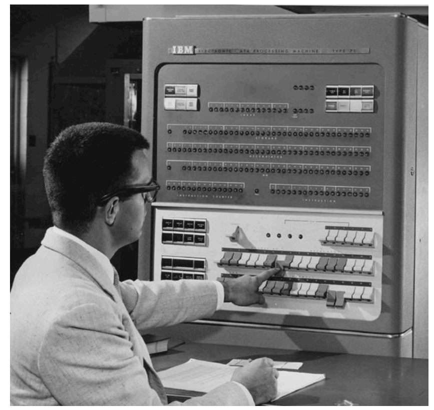
Low Latency Function Memory

Function-Function Networking

Cloud Providers as Commodity

IBM 701
1952

recode

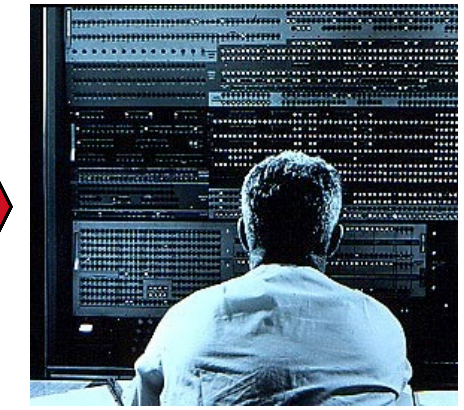


IBM 704
1954

recode

IBM 1620
1959

recode



IBM Stretch
1960

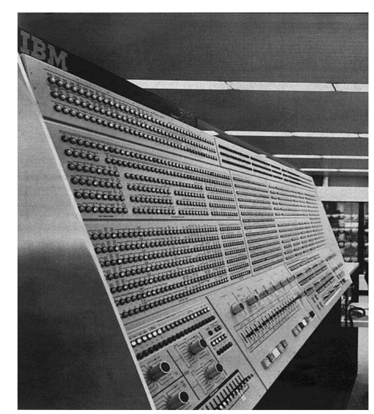
1964: Invention of the Instruction Set Architecture

IBM 360/30
1964

copy

IBM 360/67
1966

copy



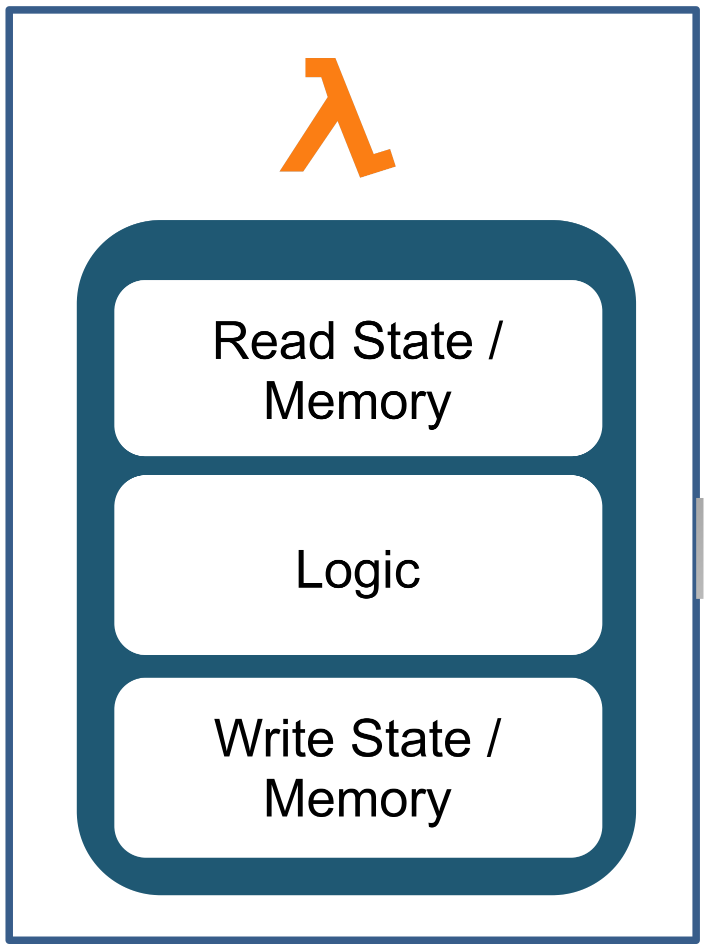
IBM 360/91
1967

copy

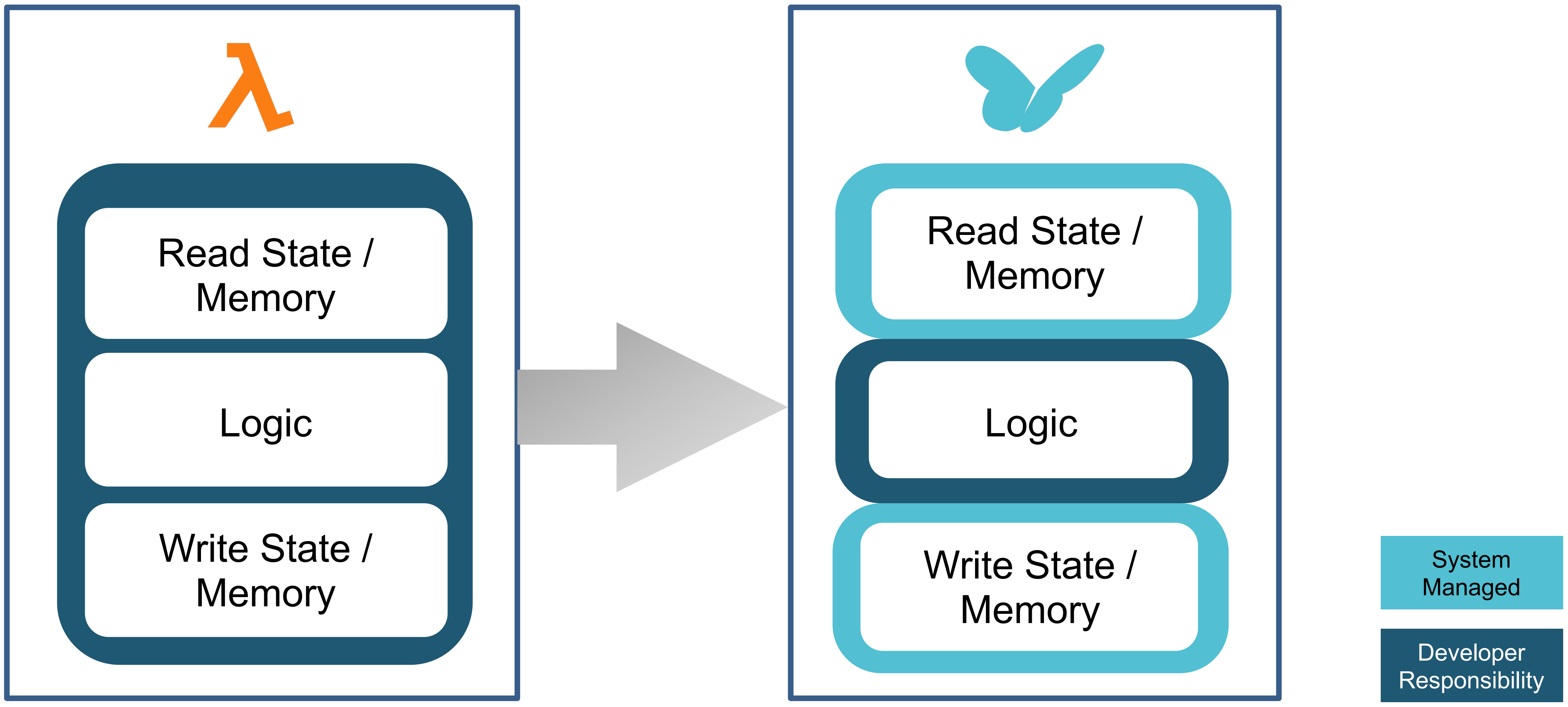
IBM 360/195
1971

Serverless & Stateful

nimbella



Developer Responsibility


Read State / Memory
Logic
Write State / Memory
Read State / Memory
Logic
Write State / Memory
System Managed
Developer Responsibility

This is hard.

functions have **transient residency**

transient residency → **no** data **locality**

☆ Formal Foundations of Serverless Computing

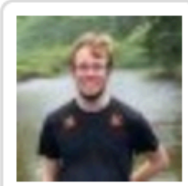
OOPSLA

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 λ_λ, an operational semantics of the essence of serverless computing. Despite being a small (half a page) core calculus, λ_λ models all the low-level details that serverless functions can observe. To show that λ_λ 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 λ_λ coincide. Second, we augment λ_λ 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 λ_λ 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
United States

Yuriy Brun
University of Massachusetts Amherst
United States

Arjun Guha
University of Massachusetts, Amherst
United States

Serverless Operational Semantics

Serverless Functions $\langle f, \Sigma, \text{recv}, \text{step}_f, \text{init} \rangle$

Functions	$F := \cdots$	
Function name	$f \in F$	
Internal states	$\Sigma := \cdots$	
Initial state	$\text{init} \in F \to \Sigma$	
Receive event	$\text{recv}_f \in v \times \Sigma \to \Sigma$	
Internal step	$\text{step}_f \in F \times \Sigma \to \Sigma \times t$	With effe
Values	$v := \cdots$	JSON, HTT
Commands	$t := \varepsilon$	
	$\mid \mathbf{return}(v)$	Return va

Serverless Functions

Values	$v := \cdots$	
	$\mid (v_1, v_2)$	Tuples
SPL expressions	$e :=$ invoke f	Invoke serverless functio
	$\mid$ first e	Run e to first part of inpu
	$\mid e_1 \ggg e_2$	Sequencing
SPL continuations	$\kappa :=$ ret x	Response to request
	$\mid$ seq e κ	In a sequence
	$\mid$ first v κ	In first
Components	$C := \cdots$	
	$\mid \mathbb{E}(e, v, \kappa)$	Running program
	$\mid \mathbb{E}(x, \kappa)$	Waiting program
	$\mid \mathbb{R}(e, x, v)$	Run program e on v

store is free

SPL expressions

$e := \cdots \mid p$ Run transformation

JSON values

$v ::= n \mid b \mid str \mid \text{null}$

JSON pattern

$p := v$ JSON literal

JSON query

q

JSON pattern

p	$:= v$	JSON literal
	$\mid [p_1, \cdots, p_n]$	Array
	$\mid \{str_1 : p_1, \cdots, str_n : p_n\}$	Object
	$\mid p_1 \; op \; p_2$	Operators
	$\mid \texttt{if} \; (p_1) \; \texttt{then} \; p_2 \; \texttt{else} \; p_3$	Conditional
	$\mid [str_1 \to p_1]$	Update field

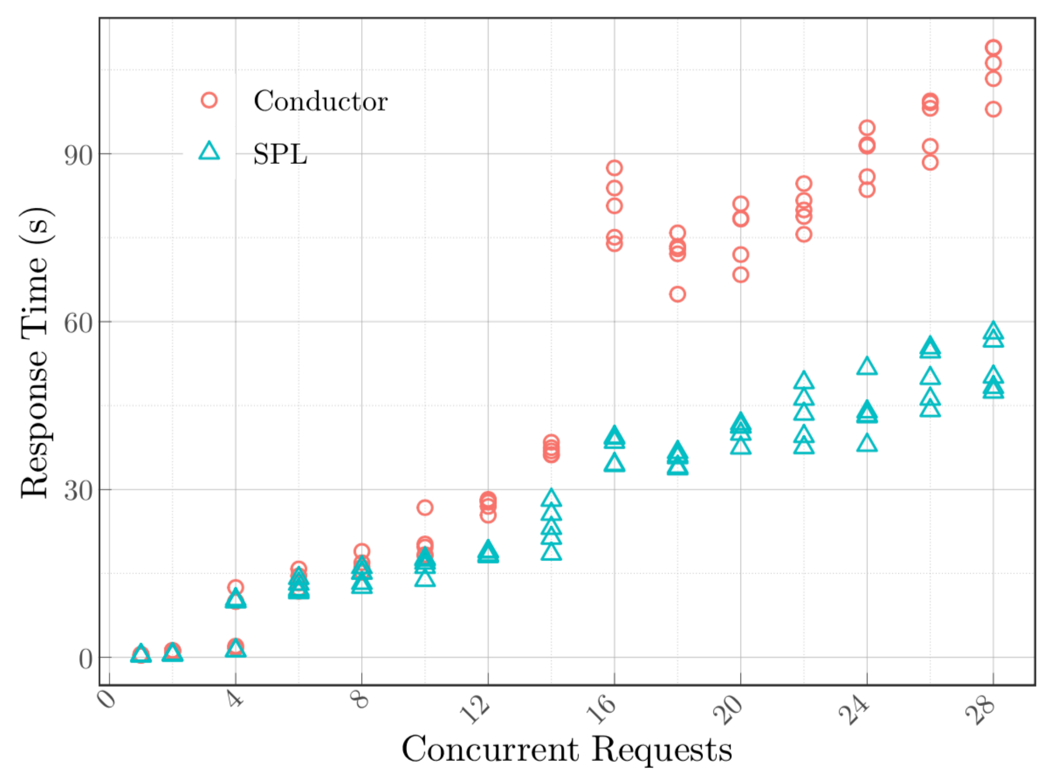
Serverless Platform

Request ID	$x := \cdots$	
Instance ID	$y := \cdots$	
Execution mode	$m := \mathbf{idle}$	Idle
	$\mid \mathbf{busy}(x)$	Processin
Transition labels	$\ell :=$	Internal
	$\mid \text{start}(f, x, v)$	Receive v
	$\mid \text{stop}(x, v)$	Respond
Components	$\mathbb{C} := \mathbb{F}(f, m, \sigma, y)$	Function
	$\mid \mathbb{R}(f, x, v)$	Apply f t
	$\mid \mathbb{S}(x, v)$	Respond
Component set	$C := \{C_1, \cdots, C_n\}$	

BeginT

EndT

DropT



"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"