SJTU SAIL (Software Architecture and Infrastructure Lab)

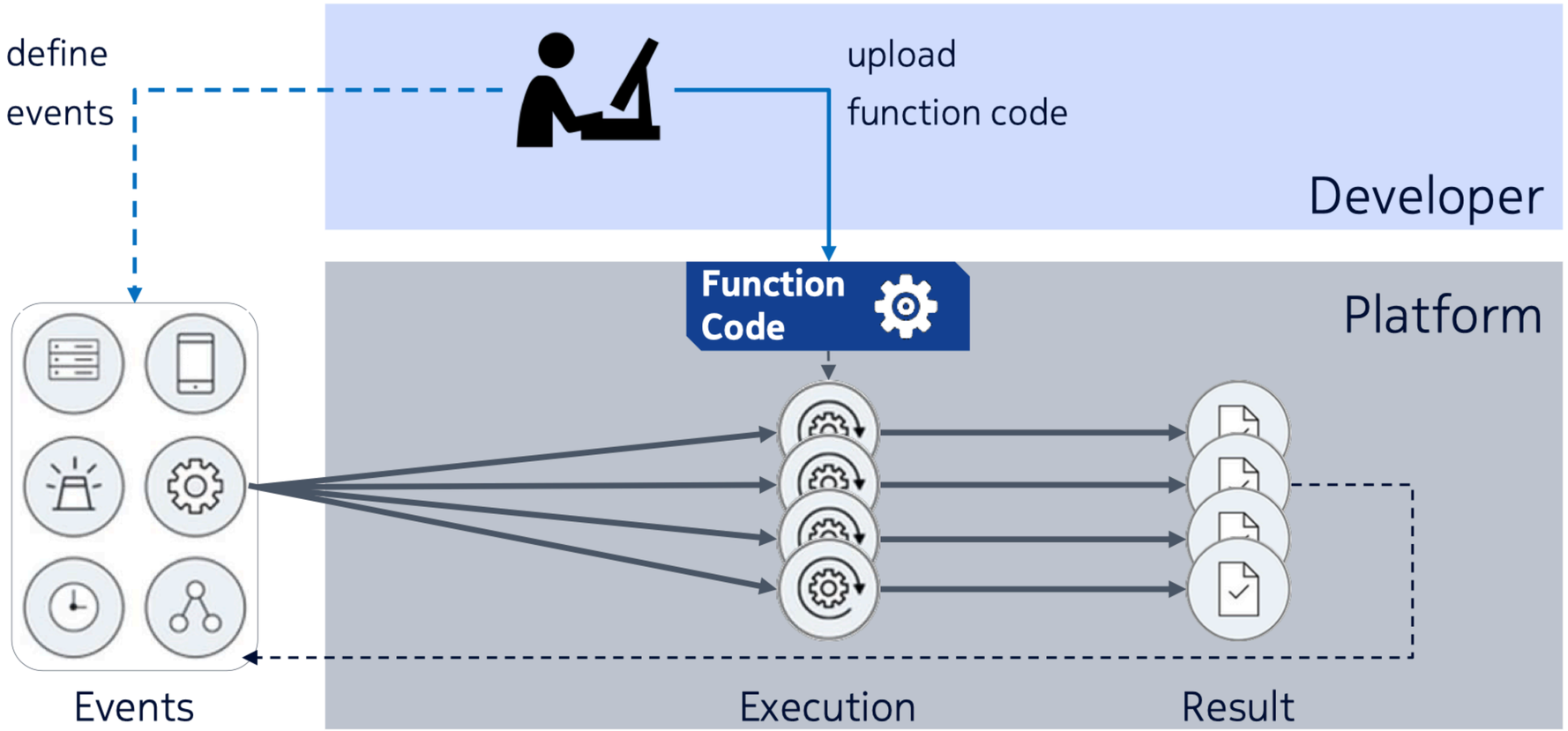
SHANGHAI JIAO TONG UNIVERSITY

Serverless

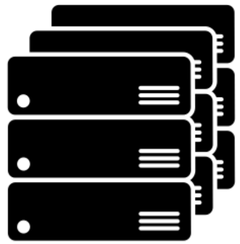
Yuchen Cheng

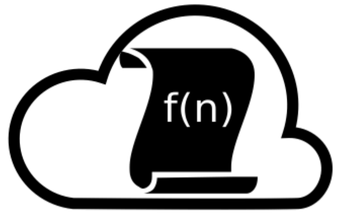
2020-09-28

Function-as-a-Service (FaaS)



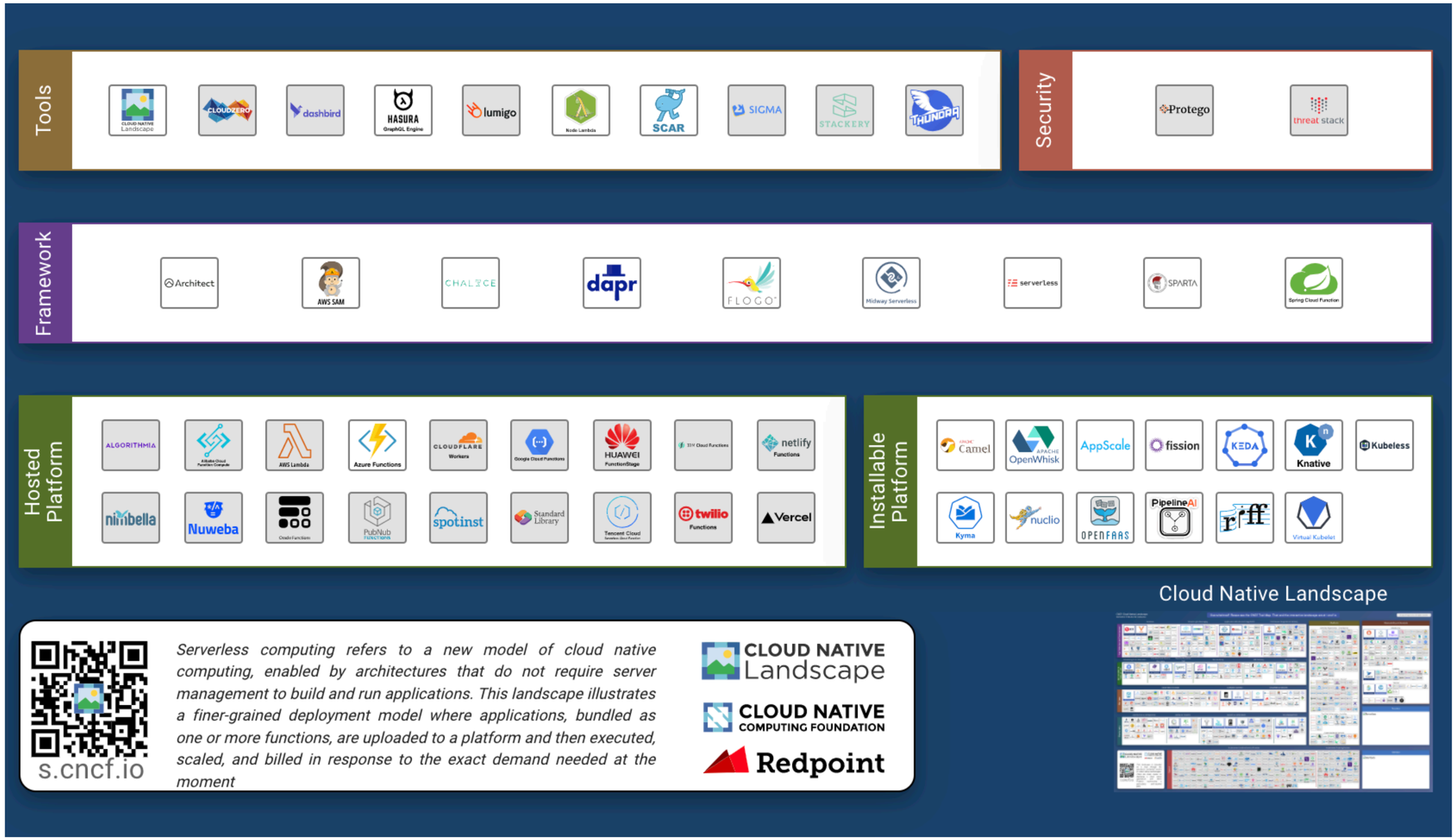
What is serverless?





	Bare Metal	VMs (IaaS)	Containers	Functions (FaaS)
Unit of Scale	Server	VM	Application/Pod	Function
Provisioning	Ops	DevOps	DevOps	Cloud Provider
Init Time	Days	~1 min	Few seconds	Few seconds
Scaling	Buy new hardware	Allocate new VMs	1 to many, auto	0 to many, auto
Typical Lifetime	Years	Hours	Minutes	O(100ms)
Payment	Per allocation	Per allocation	Per allocation	Per use
State	Anywhere	Anywhere	Anywhere	Elsewhere

CNCF cloud native landscape for serverless



Serverless

Serverless Computing: One Step Forward, Two Steps Back

Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov and Chenggang Wu
UC Berkeley
{hellerstein,jmfaleiro,jegonzal,jssmith,vikrams,atumanov,...

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

Cloud Programming Simplified: A Berkeley View on Serverless Computing

Eric Jonas, Anurag Khandelwal, Karl Krauth, Johann Schleier-Smith, Qifan Pu, Neeraja Yadwadkar, Ion Stoica, Vikram Sreekanti, Vaishaal Shankar, Joseph E. Gonzalez, David A. Patterson, Chia-Che Tsai, Joao Carreira, Raluca Ada Popa

UC Berkeley

serverlessview@berkeley.edu

"... we predict that (...) serverless computing will grow to dominate the future of cloud computing."

SAND: Towards High-Performance Serverless Computing

Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt, Nokia Bell Labs

Background

Overview of existing platforms

- Functions are isolated with containers
- Containers are deployed where resources are available
- Containers handle events and stay deployed until a timeout
- Functions interact via a distributed message bus

Implications of common practices

Implications of Common Practices

Function Container

Function execution & concurrency:

1. Start a new container for every function execution (i.e., cold start) → long invocation latency
2. Keep and reuse idle containers (i.e., warm start)
3. Concurrency: cold starts or queuing → resource inefficiency

Function interaction:

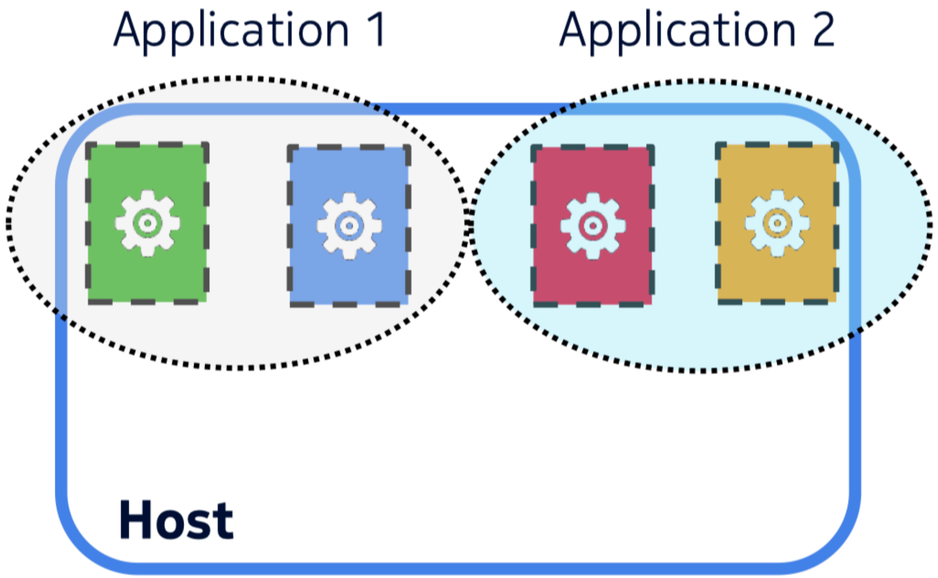
- Go through the distributed message bus → long function interaction latency

Key Ideas and Building Blocks

Application sandboxing

Insight: Different concepts should have different fault isolation

- Stronger isolation between applications
- Weaker isolation between functions of the same application



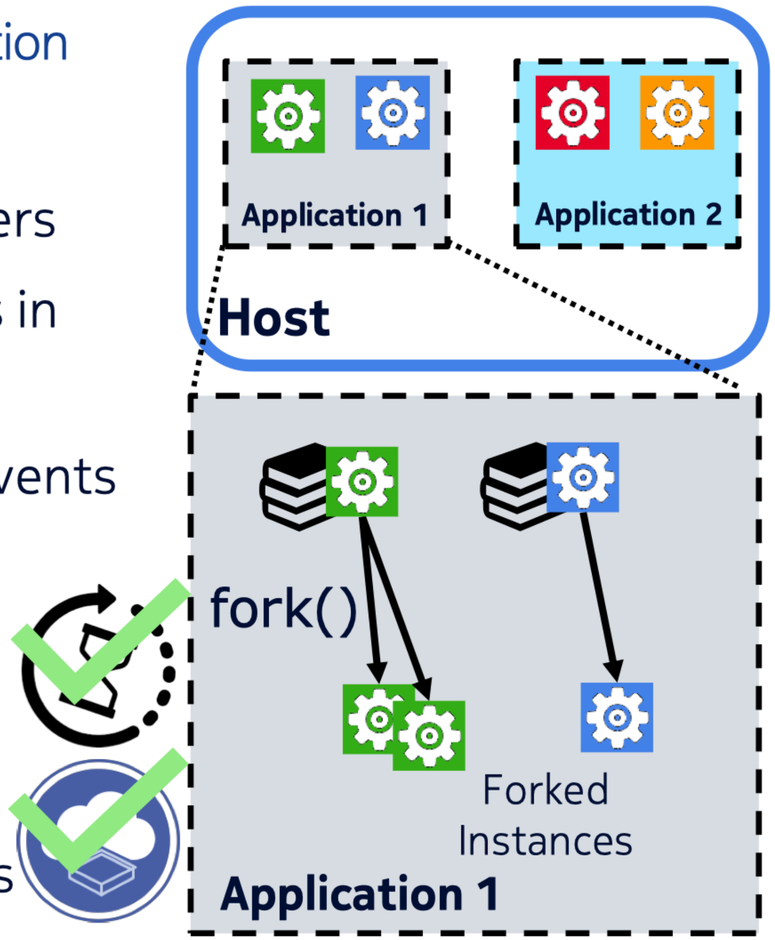
Application sandboxing operation

SAND Application-level Sandboxing Operation

1) Put applications in separate containers
2) Run functions as separate processes in the same container
3) Fork new processes to handle new events

Advantages:

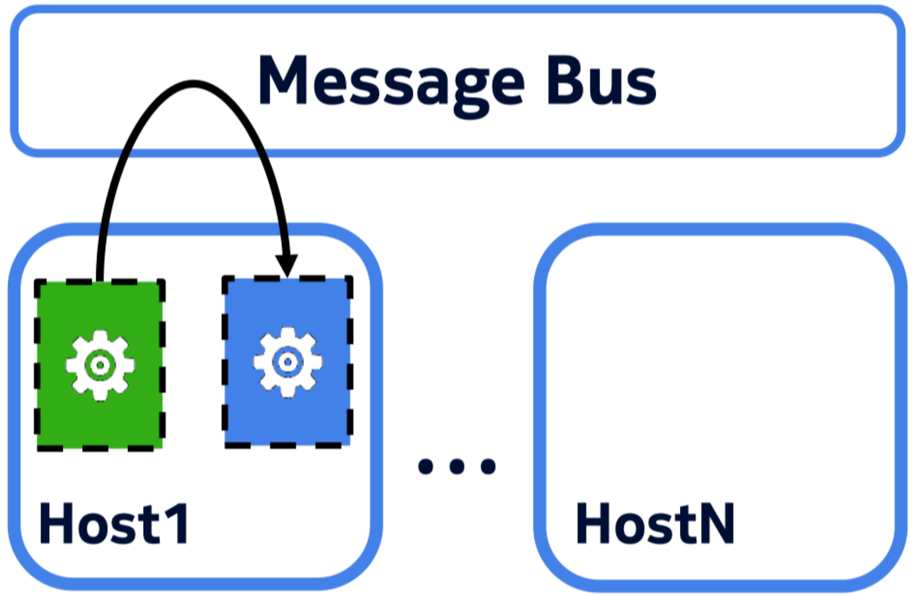
1) Fast creation of function executions
2) Low execution footprint
3) Automatic de-allocation of resources



Hierarchical message queuing

Insight: Exploit locality of the functions

- Shortcuts for interacting functions of an application

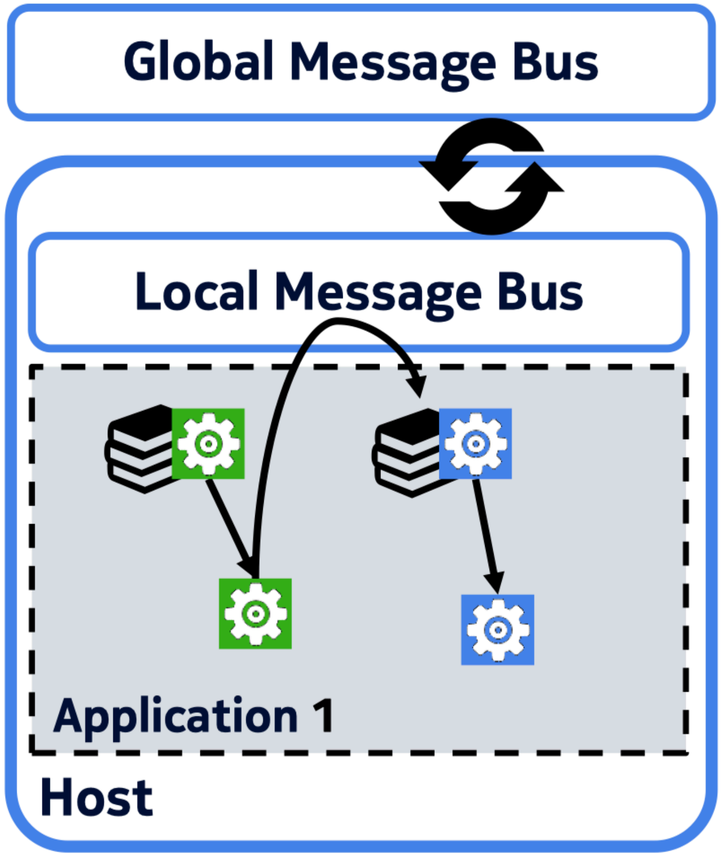


Hierarchical message queuing operation

1) Run a local message bus on each host
2) Functions interact with other functions via the local message bus
3) Coordinate local bus with the global bus

Advantages:

1) Low function interaction latency
2) Fault tolerance & parallelism if needed



Workflow example

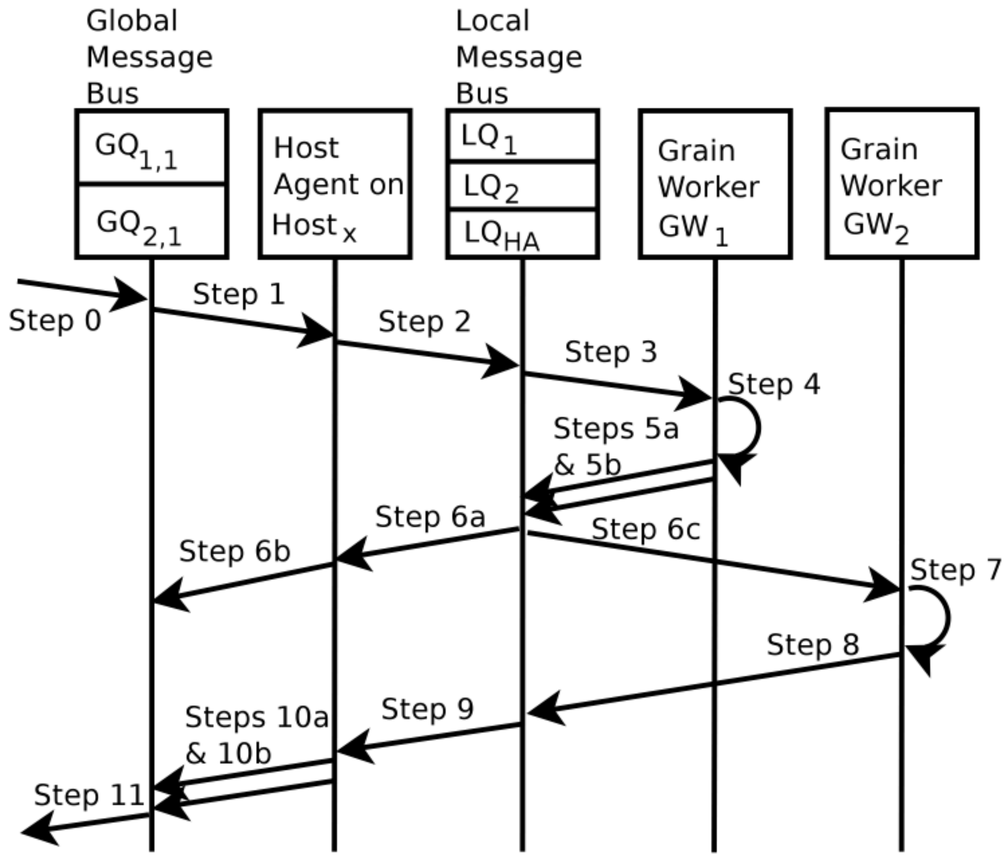
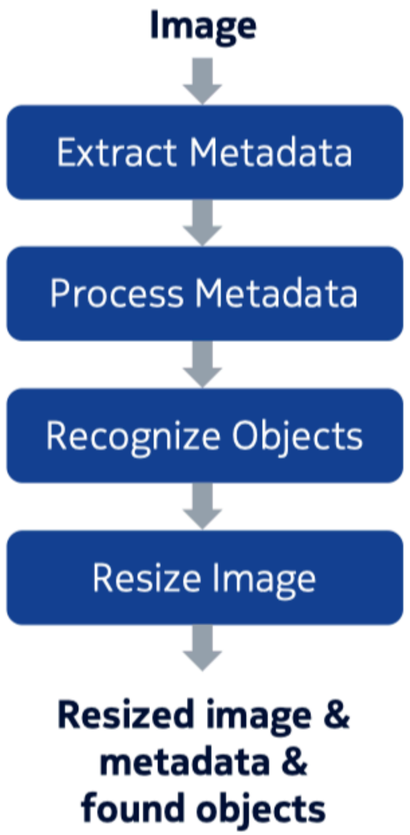
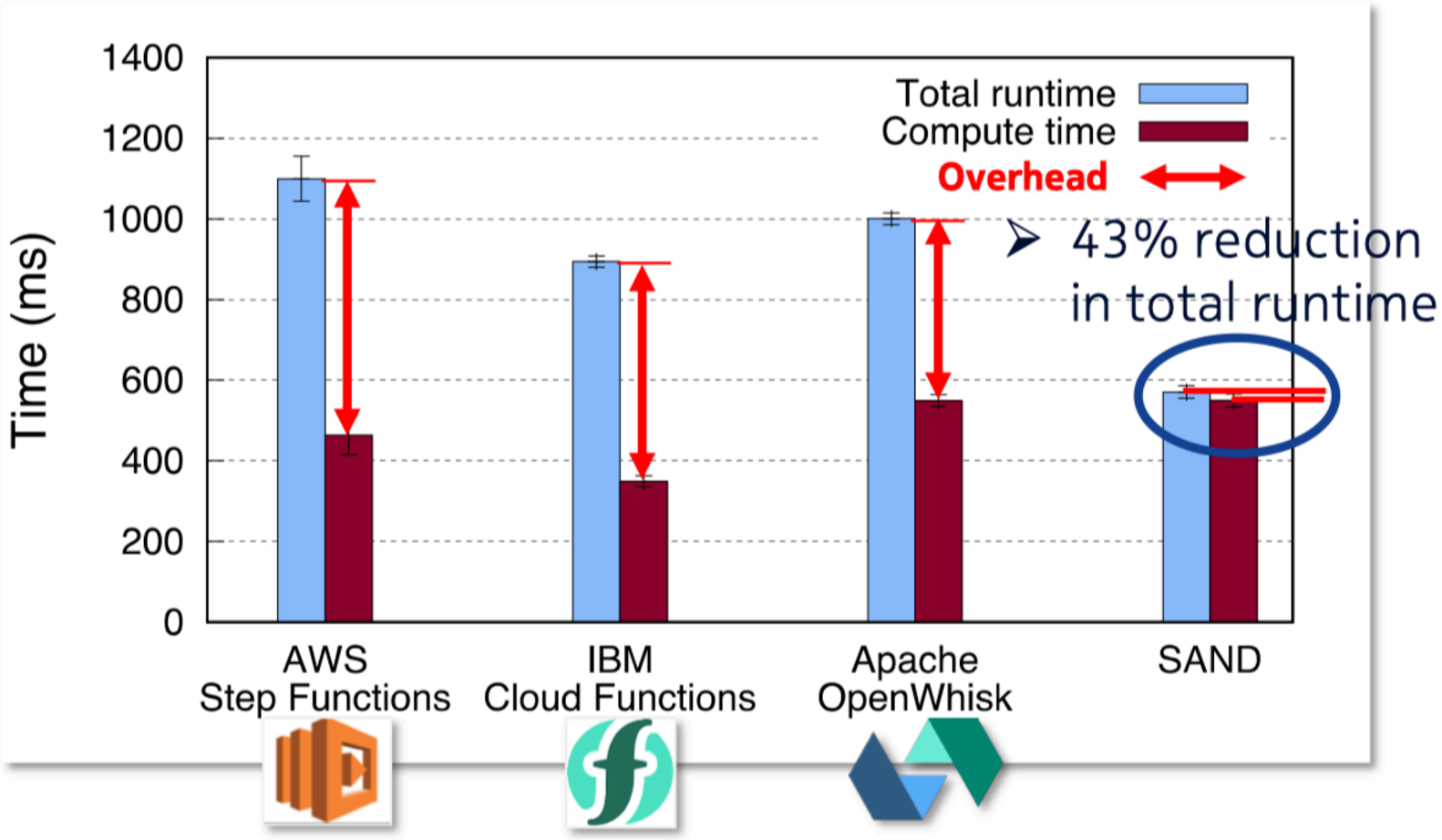


Figure 4: Handling of a user request to a simple application that consists of two grains in a workflow.

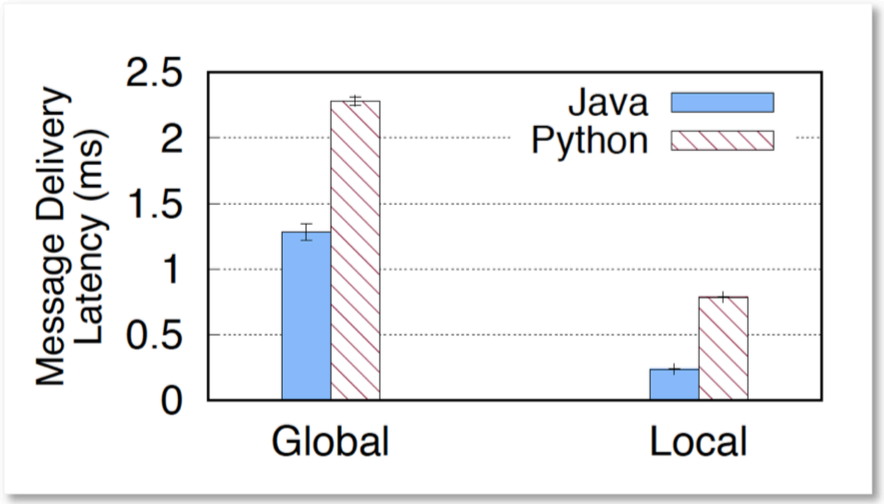
Evaluation

Overhead Comparison

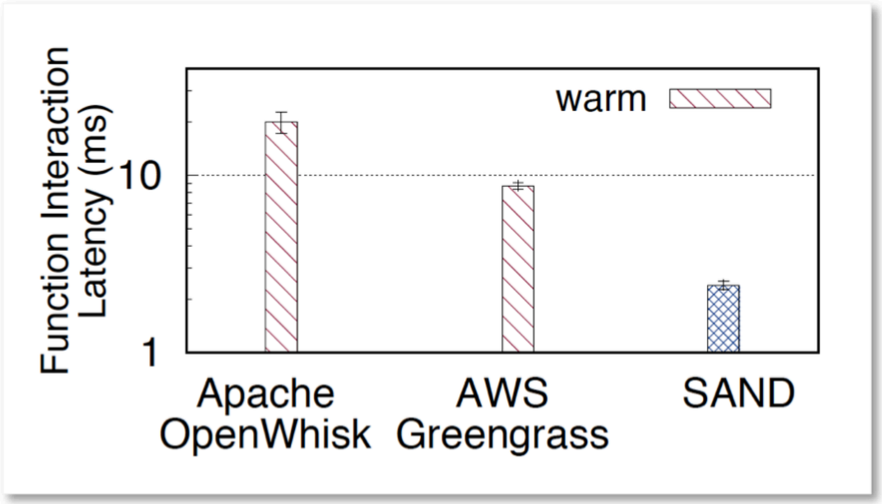




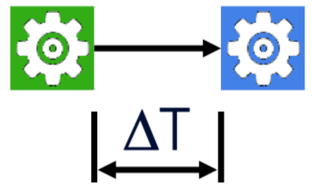
Microbenchmarks



- Access to local bus is 3-5x faster than global bus



- 8.3x as fast as OpenWhisk
- 3.6x as fast as Greengrass



SAND: Summary

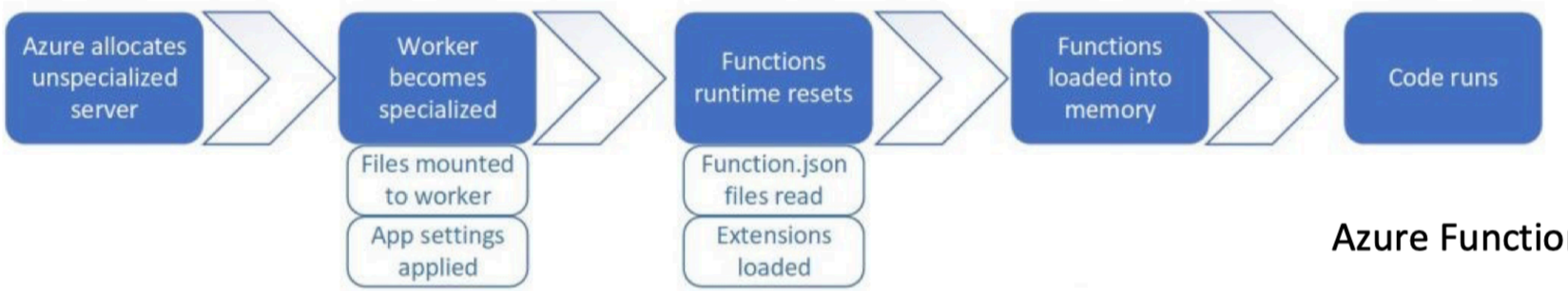
- A fine-grained application sandboxing mechanism for serverless computing
  - isolation between different applications
  - isolation between functions of the same application
- A hierarchical message queuing and storage mechanism to leverage locality for the interacting functions of the same application
  - a local message bus on each host to create shortcuts to enable fast message passing between interacting functions
  - a global message bus that serves as a backup of locally produced and consumed messages in case of failures

Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider

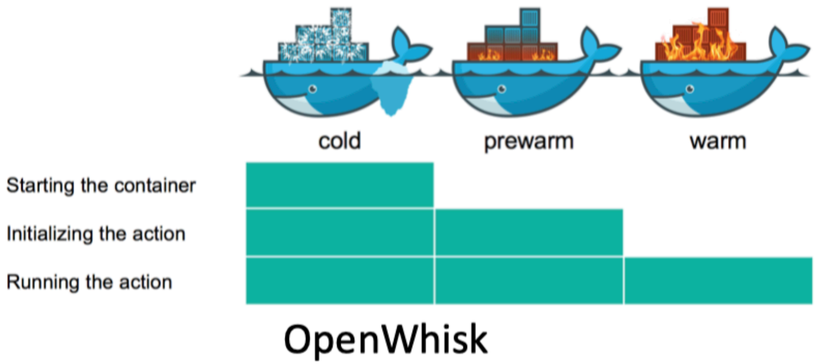
Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini, Microsoft Azure and Microsoft Research

Background

Cold starts



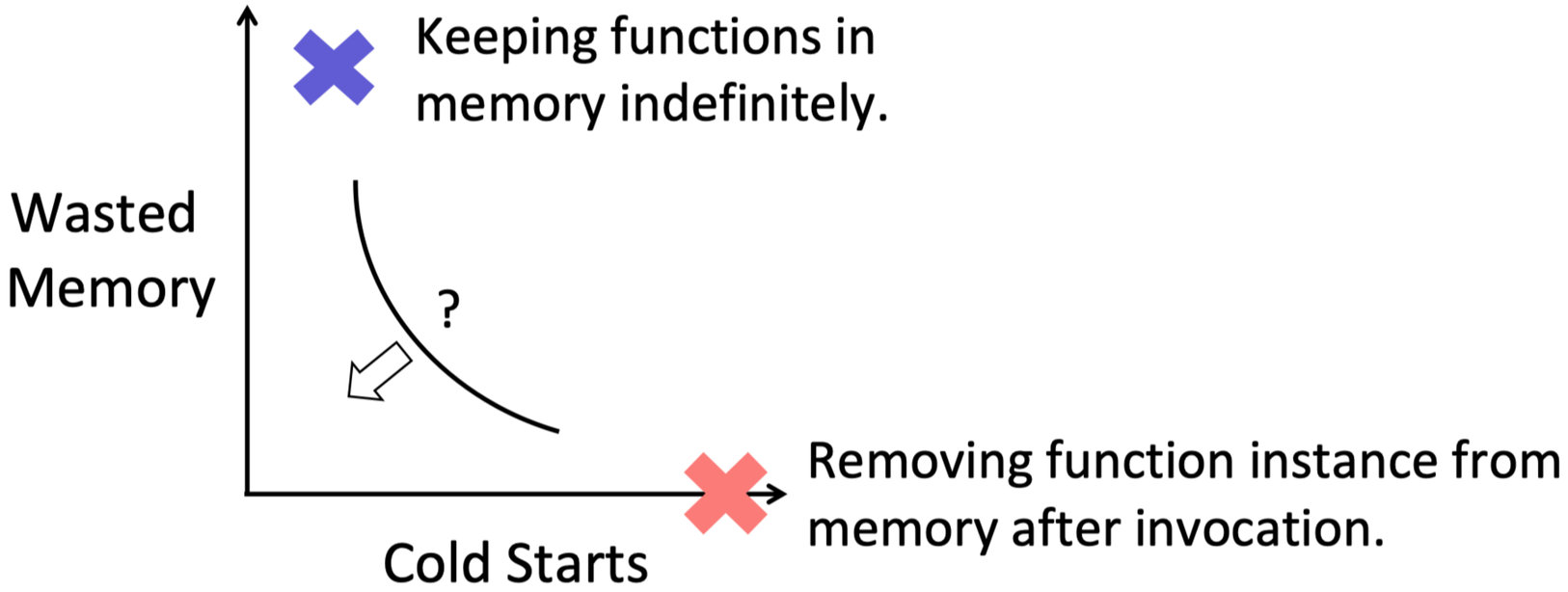
Azure Functions



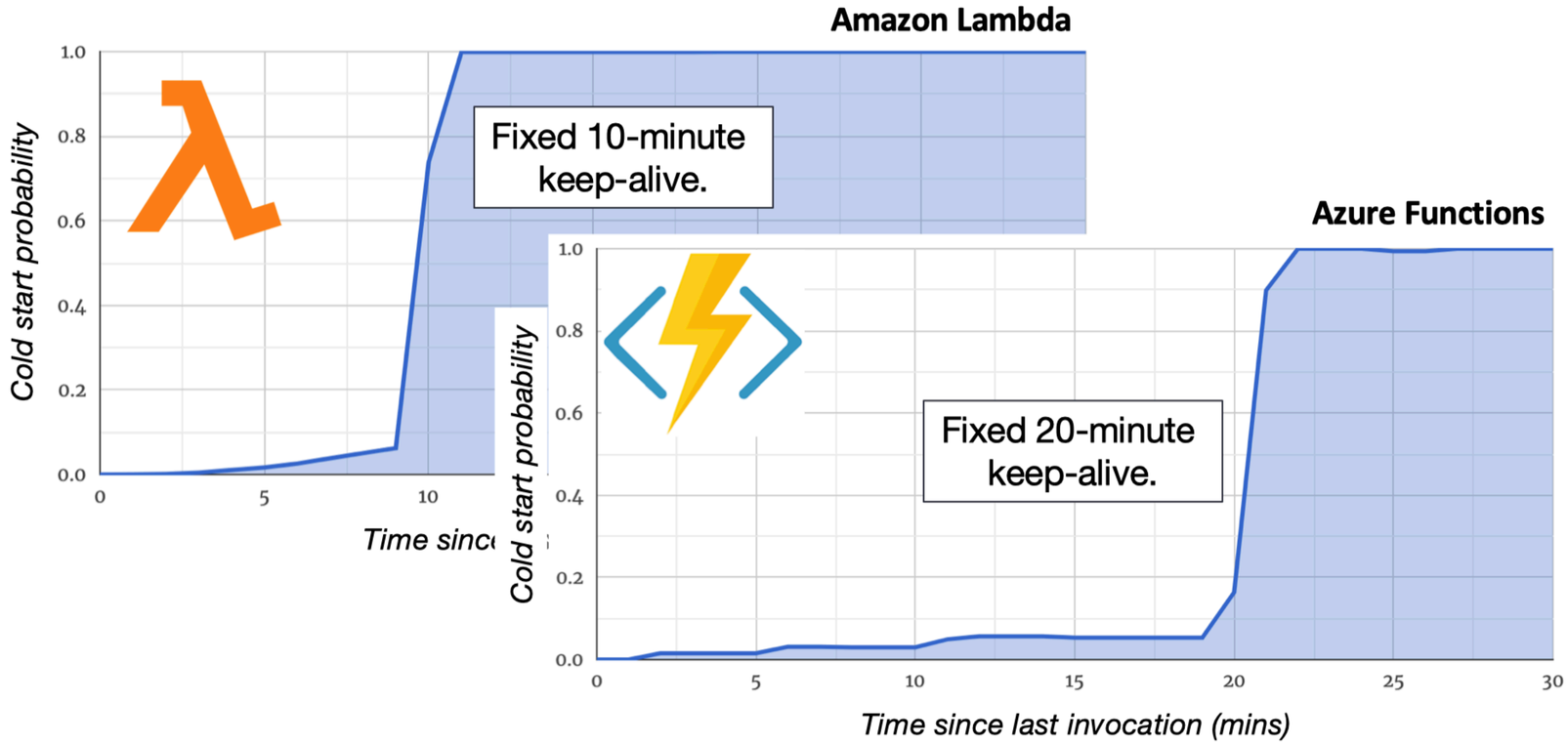
OpenWhisk

AWS Lambda

Cold starts and resource wastage



What do serverless providers do?



Mikhail Shilkov, Cold Starts in Serverless Functions, https://mikhail.io/serverless/coldstarts/

FaaS Workloads

Stepping back: Characterizing the workload

- How are functions accessed
- What resources do they use
- How long do functions take

2 weeks of all invocations to Azure Functions in July 2019 (July 15th to July 28th)

First characterization of the workload of a large serverless provider

https://github.com/Azure/AzurePublicDataset

Functions and applications

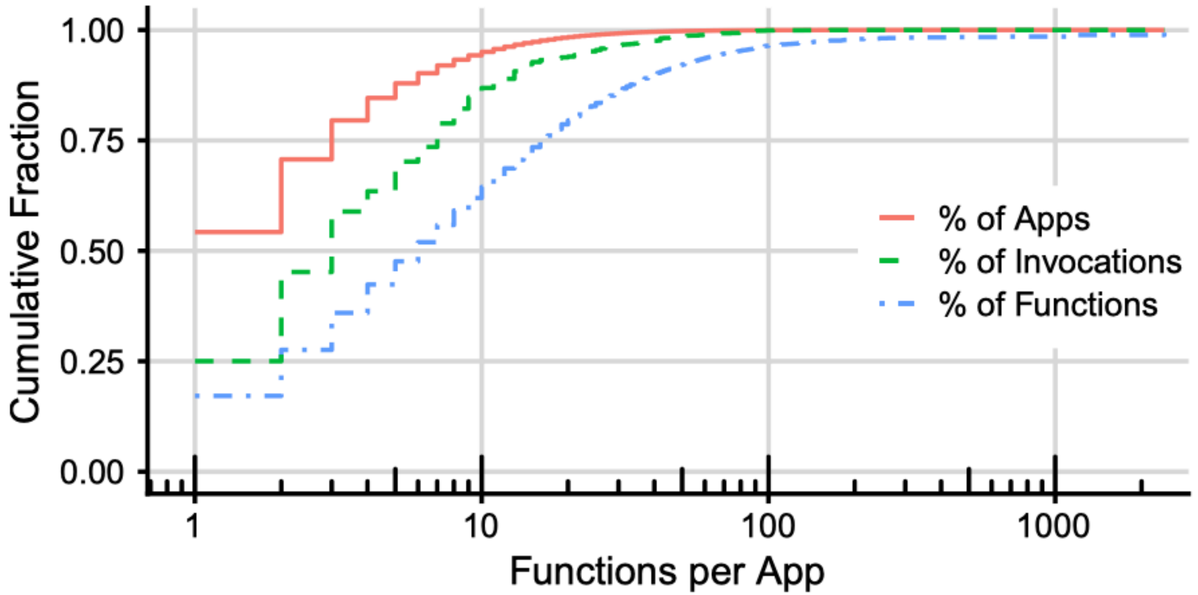


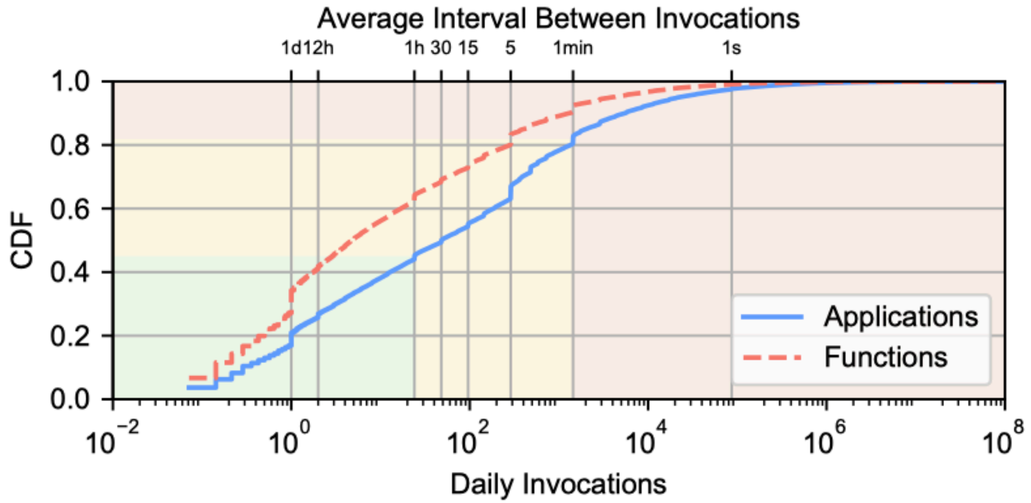
Figure 1: Distribution of the number of functions per app.

Triggers and applications

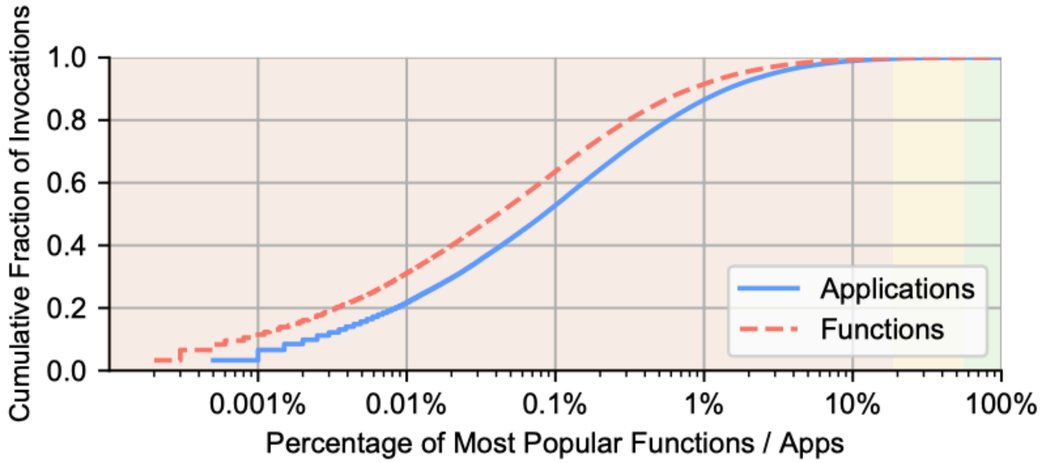
Trigger	%Functions	%Invocations
HTTP	55.0	35.9
Queue	15.2	33.5
Event	2.2	24.7
Orchestration	6.9	2.3
Timer	15.6	2.0
Storage	2.8	0.7
Others	2.2	1.0

Figure 2: Functions and invocations per trigger type.

Invocation patterns

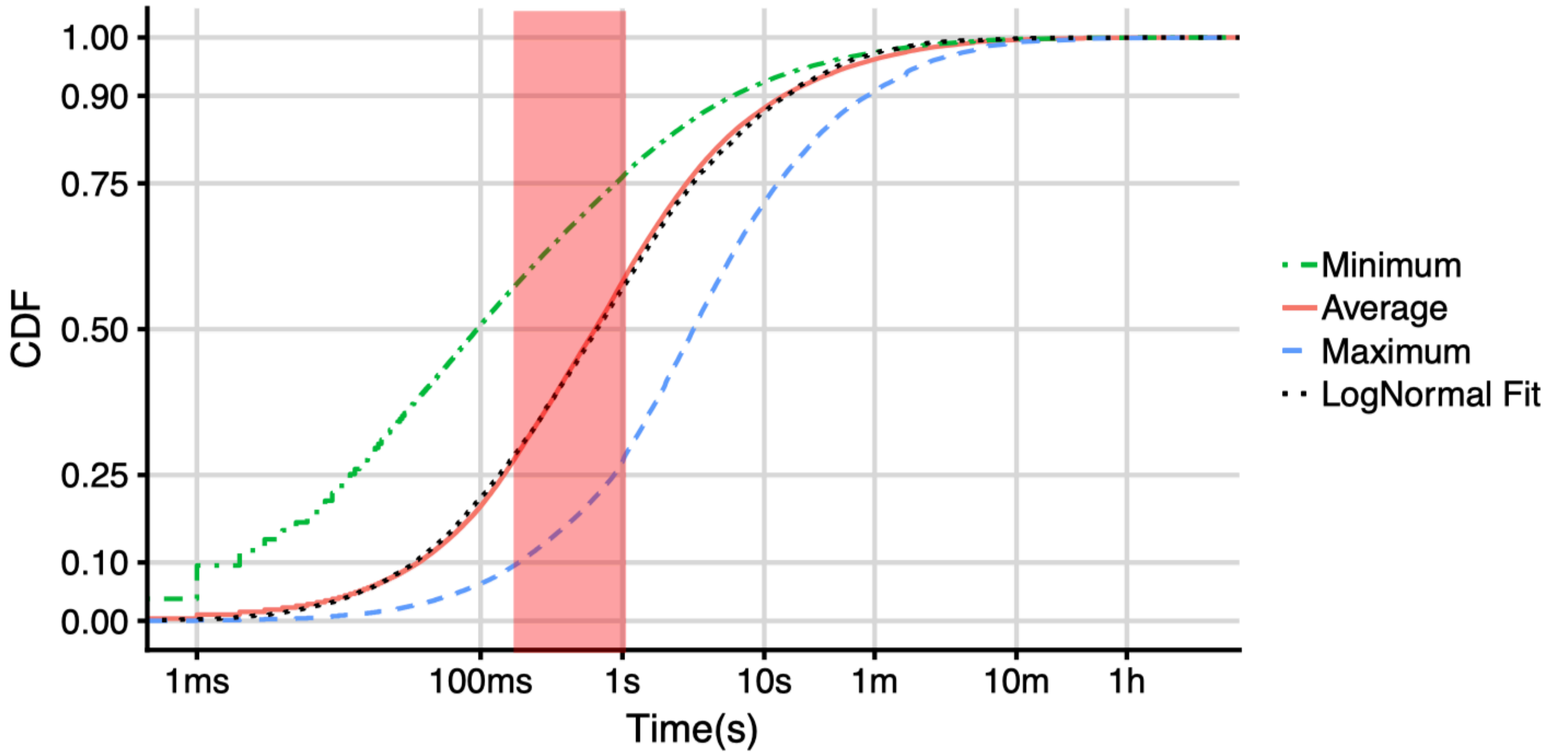


(a) CDF of daily invocations per function and application, and the corresponding *average* interval between invocations. Shaded regions show applications invoked on average at most once per hour (green, 45% of apps) and at most once per minute (yellow, 81% of apps).



(b) Fraction of total function invocations by the fraction of the most popular functions and applications. Same colors as in Figure 5(a).

Function execution times



Memory usage

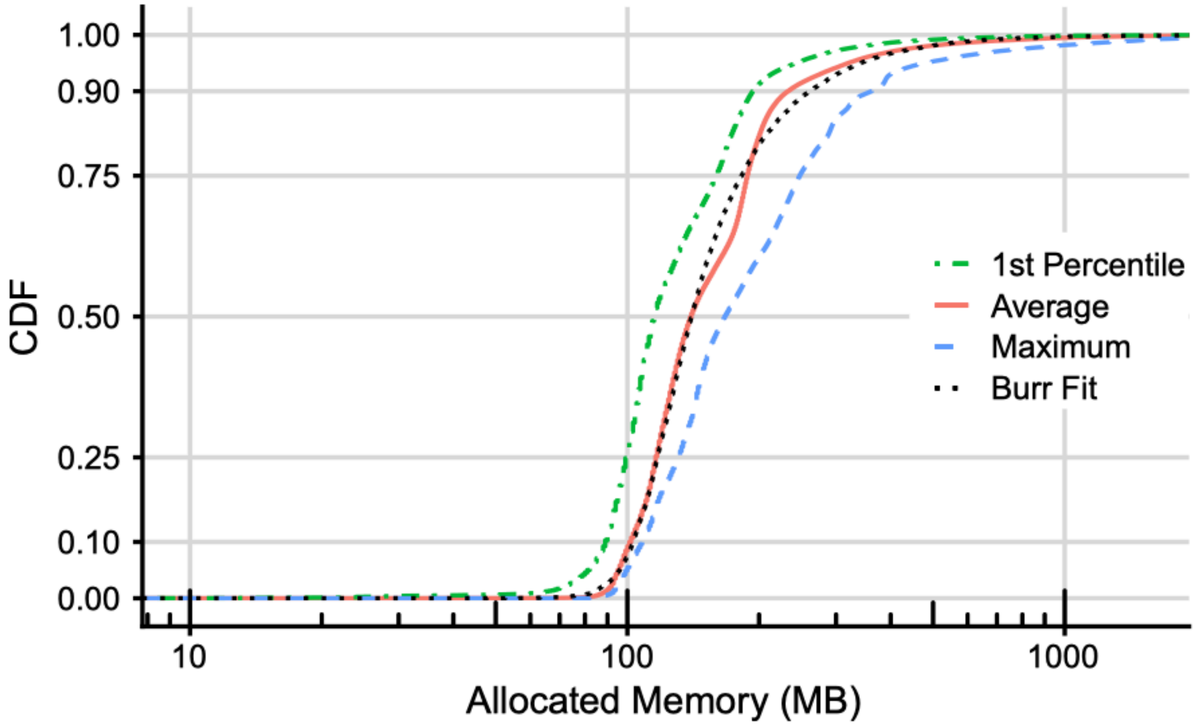


Figure 8: Distribution of allocated memory per application.

Main takeaways

- The vast majority of functions execute on the order of a few seconds
  - 75% of them have a maximum execution time of 10 seconds
- The vast majority of applications are invoked infrequently
  - 81% of them average at most one invocation per minute
- Many applications show wide variability in their IATs
  - 40% of them have a CV of their IATs higher than 1

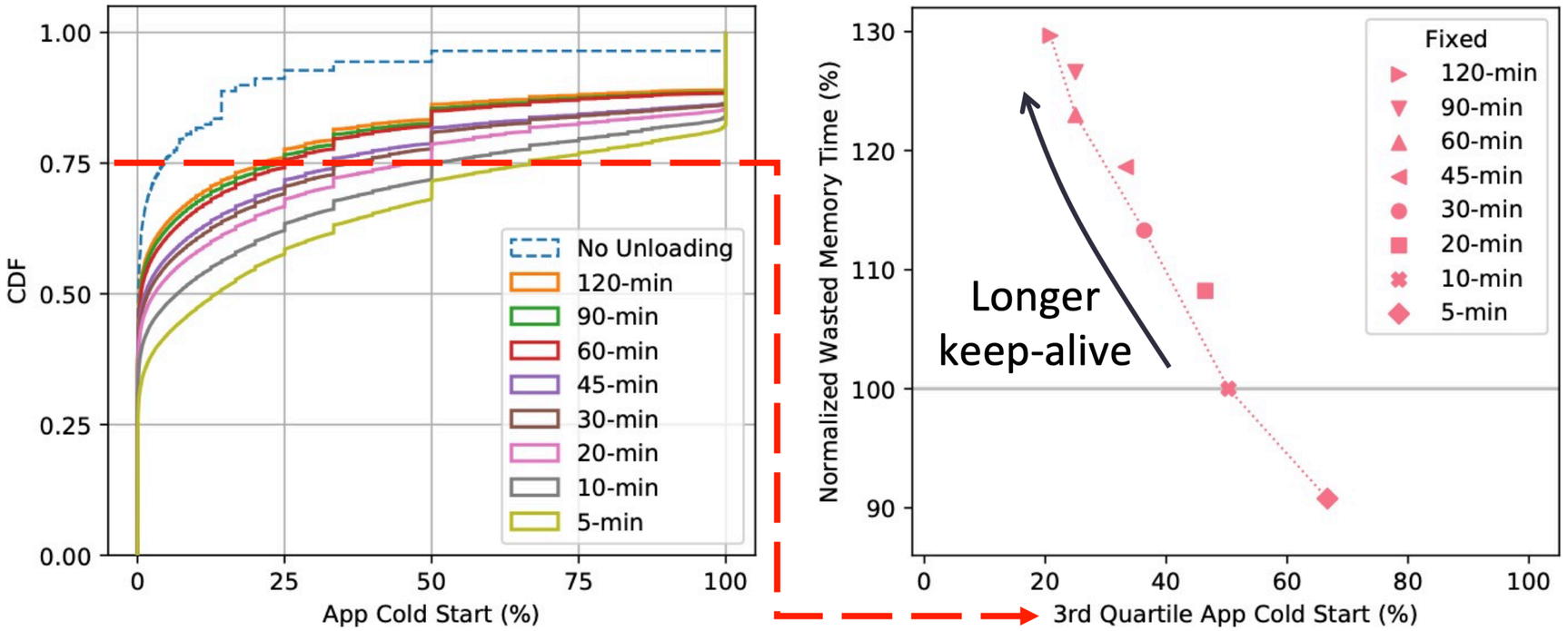
Managing Cold Starts in FaaS

Parameters

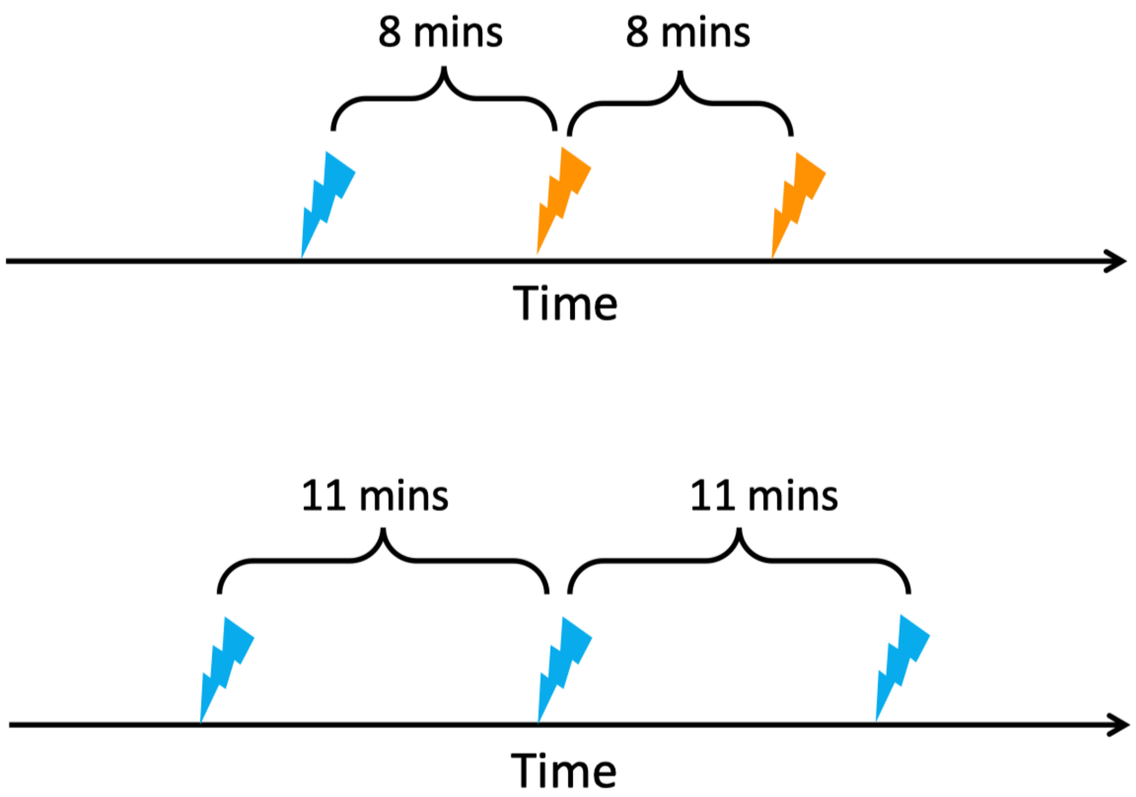
- **Pre-warming** window
  - the time the policy waits, since the last execution, before it loads the application image expecting the next invocation
- **Keep-alive** window
  - the time during which an application's image is kept alive after
    - it has been loaded to memory (pre-warming window ≥ 0) or
    - a function execution (pre-warming window = 0)

Fixed keep-alive policy

Results from simulation of the entire workload for a week.



Fixed keep-alive won’t fit all

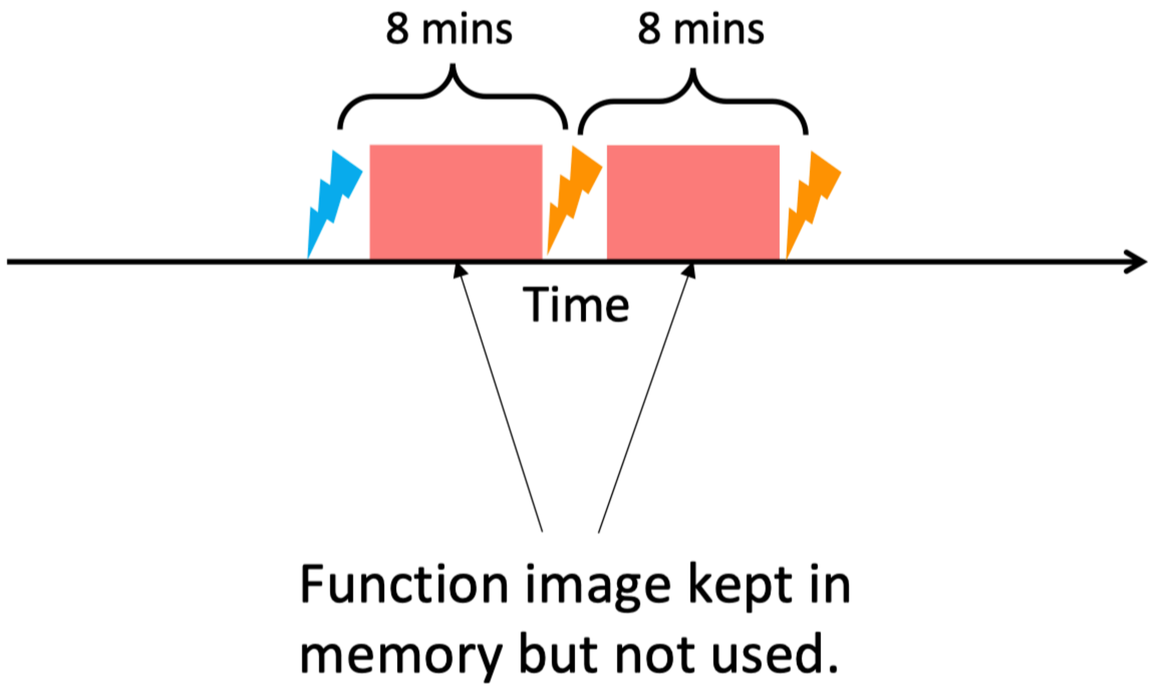


10-minute
Fixed
Keep-alive

Cold Start

Warm Start

Fixed keep-alive is wasteful



10-minute Fixed Keep-alive

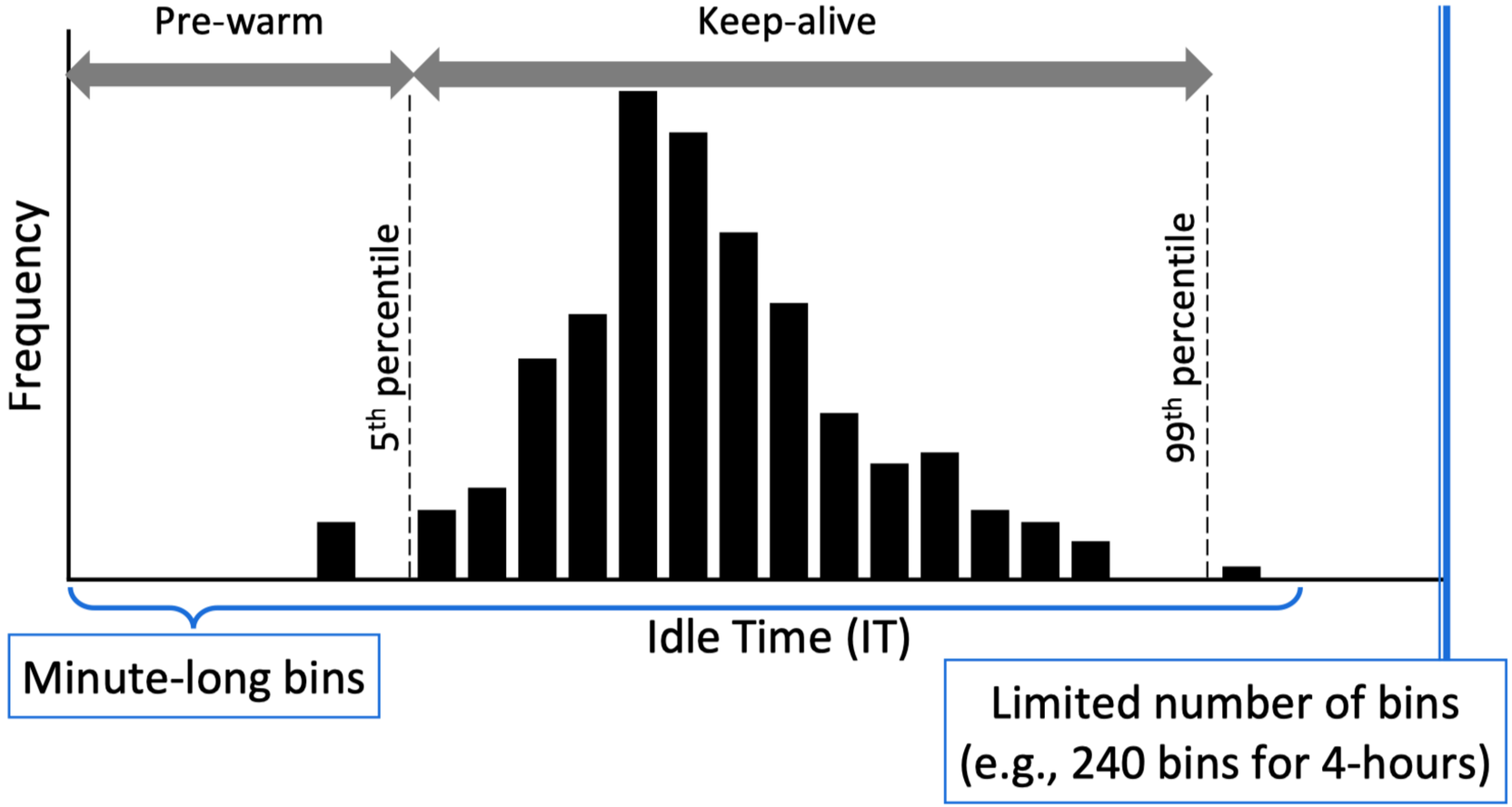
Cold Start

Warm Start

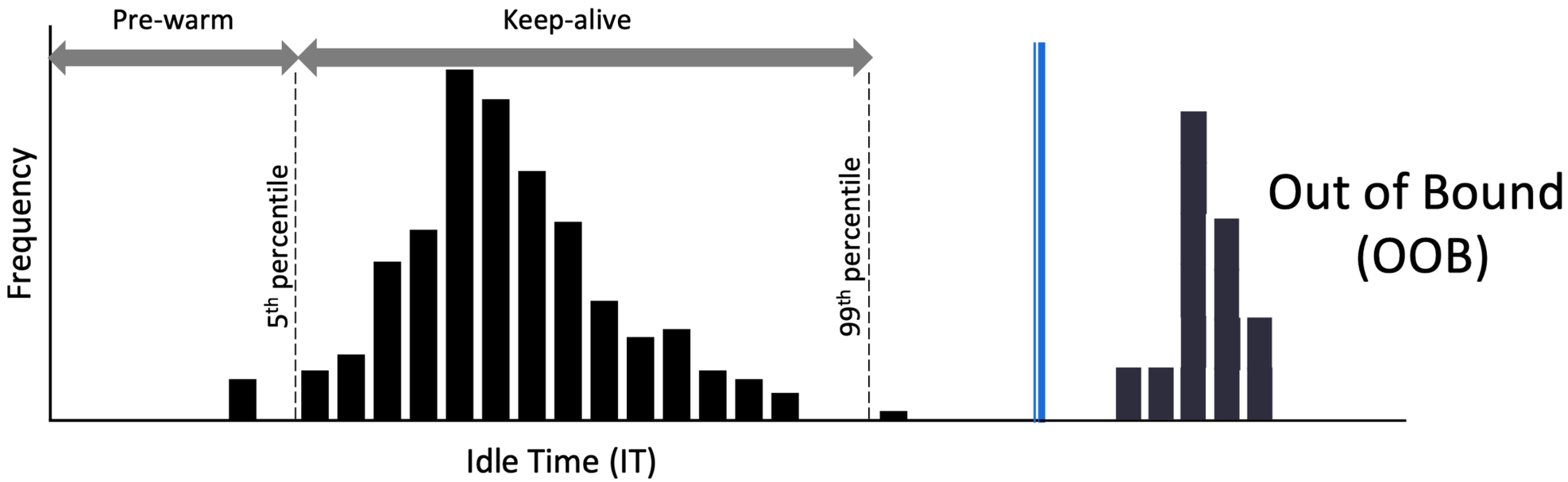
Design Challenges

1. Hard-to-predict invocations
2. Heterogeneous applications
3. Applications with infrequent invocations
4. Tracking overhead
5. Execution overhead

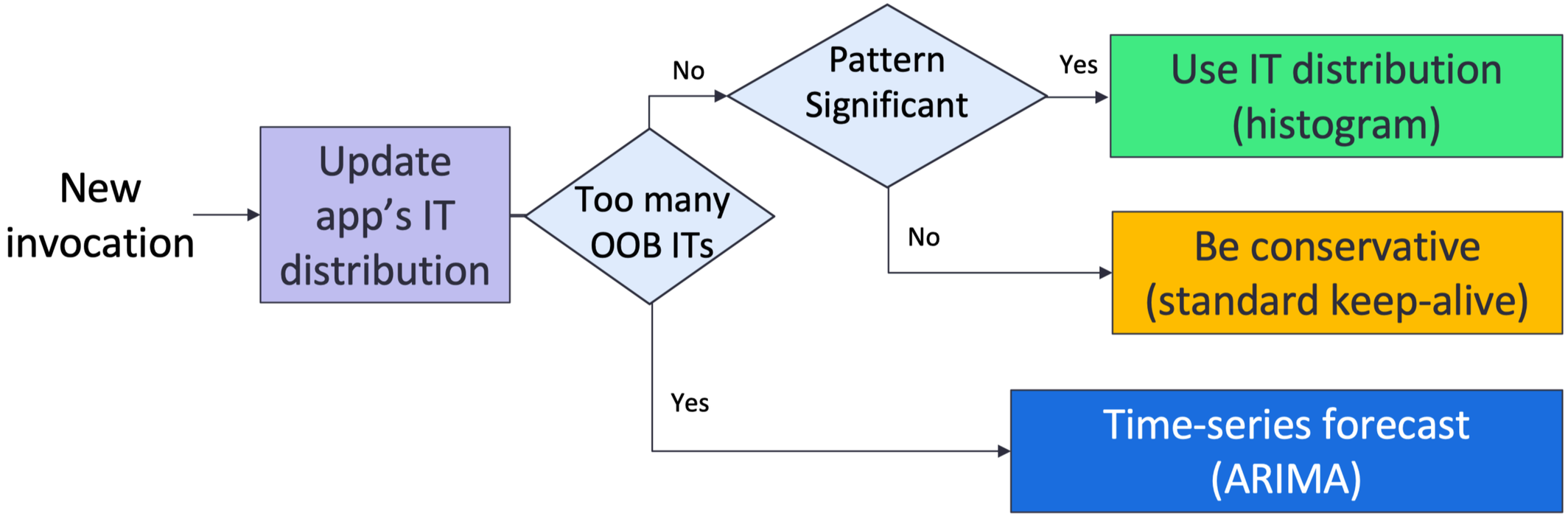
A histogram policy to learn idle times



Hybrid histogram policy



Hybrid histogram policy



Evaluation

More optimal Pareto frontier

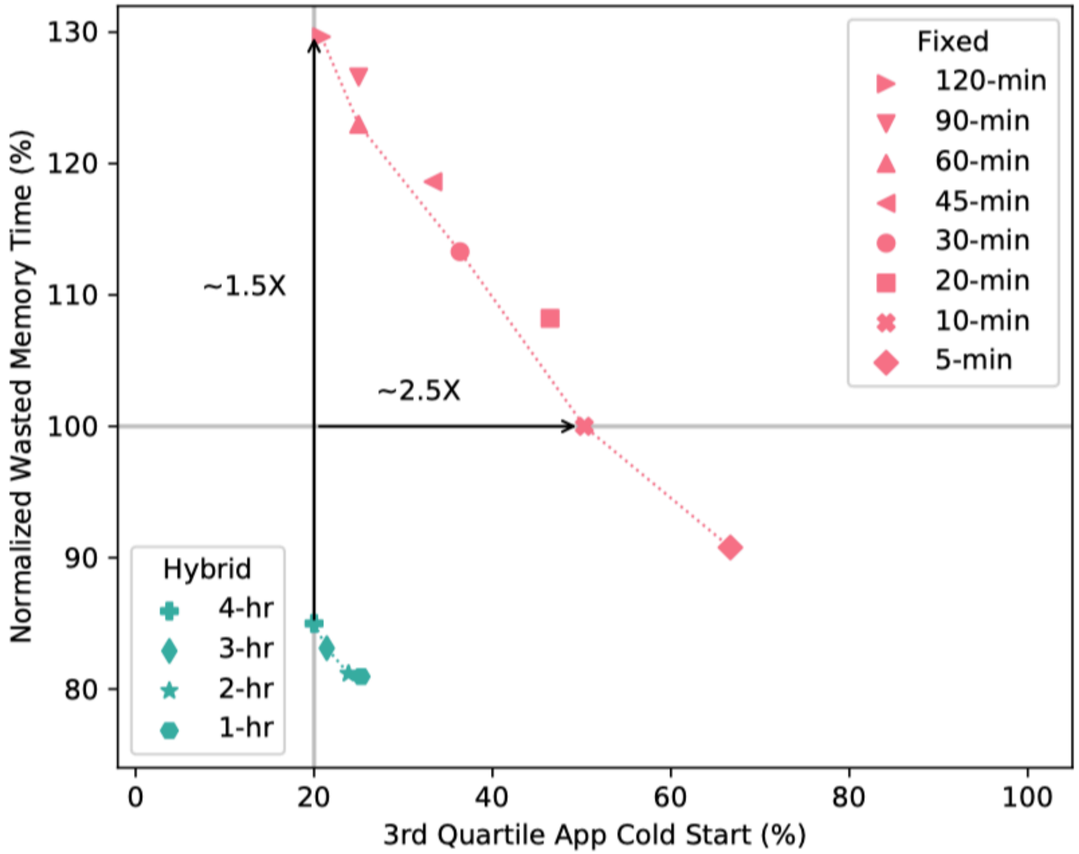
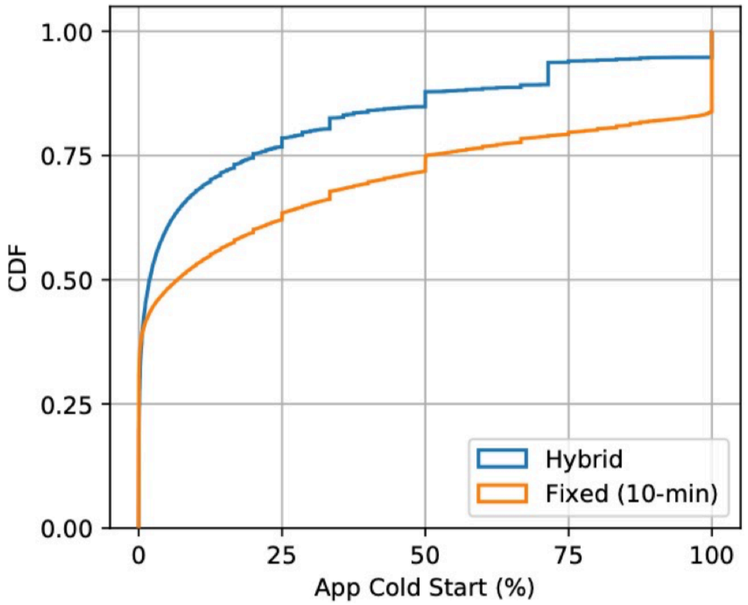


Figure 15: Trade-off between cold starts and wasted memory time for the fixed keep-alive policy and our hybrid policy.

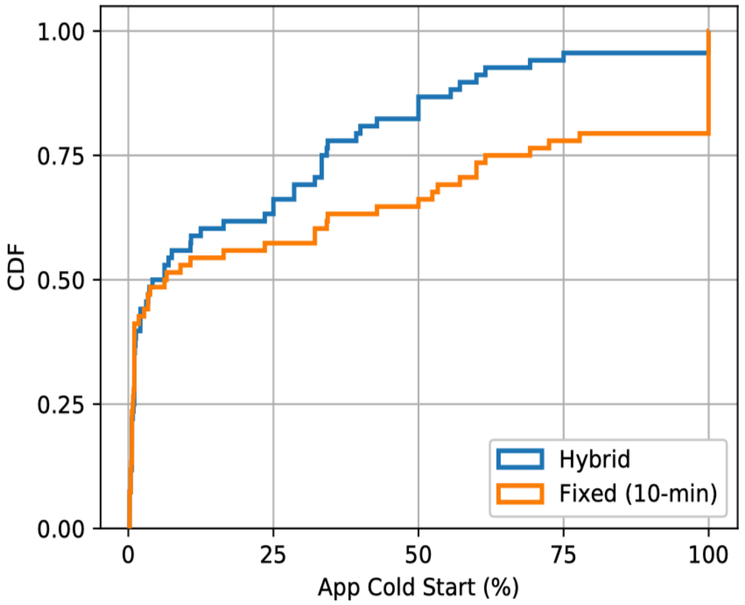
Experimental results

4-Hour Hybrid Histogram

Simulation



Experimental



Average exec time reduction: 32.5%

99th–percentile exec time reduction: 82.4%

Container memory reduction: 15.6%

Latency overhead: < 1ms (835.7µs)

Summary

- A detailed characterization of the entire production FaaS workload at a large cloud provider
- A new policy for reducing the number of cold start function executions at a low resource provisioning cost
- Extensive simulation and experimental results based on real traces showing the benefits of the policy
- An overview of our implementation in Azure Functions
- A large sanitized dataset containing production FaaS traces

Thanks for your attention.