Fork in the Road: Reflections and Optimizations for Cold Start Latency in Production Serverless Systems

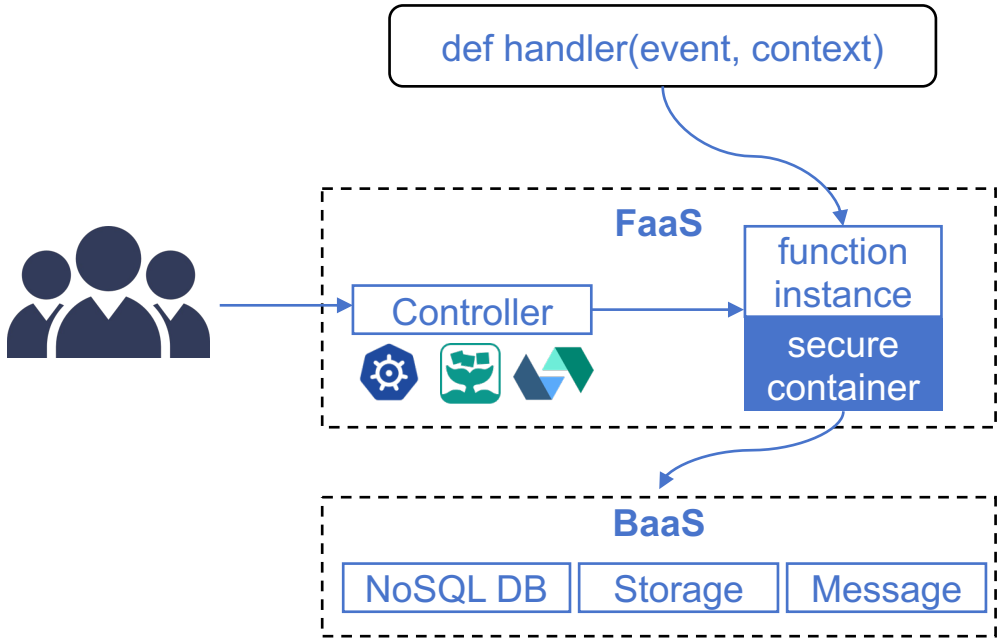
Xiaohu Chai[1,2], Tianyu Zhou[2] , Keyang Hu[1] , Jianfeng Tan[2] , Tiwei Bie[2] , Anqi Shen[2] , Dawei Shen[2] , Qi Xing[2] , Shun Song[2] , Tongkai Yang[2] , Le Gao[2] , Feng Yu[2] , Zhengyu He[2] , Dong Du[3] , Yubin Xia[3] , Kang Chen[1] , and Yu Chen[4,1]

[1]Tsinghua University, [2]Ant Group, [3]Shanghai Jiao Tong University, [4]Quan Cheng Laboratory

蚂蚁集团
ANT GROUP

Serverless: Wide Adoption in Public and Private Clouds

- **Benefits**
  - On-Demand Scaling
  - Pay-As-You-Go Pricing
  - Focus on Code
  - …



AWS Lambda

Microsoft Azure

Meta

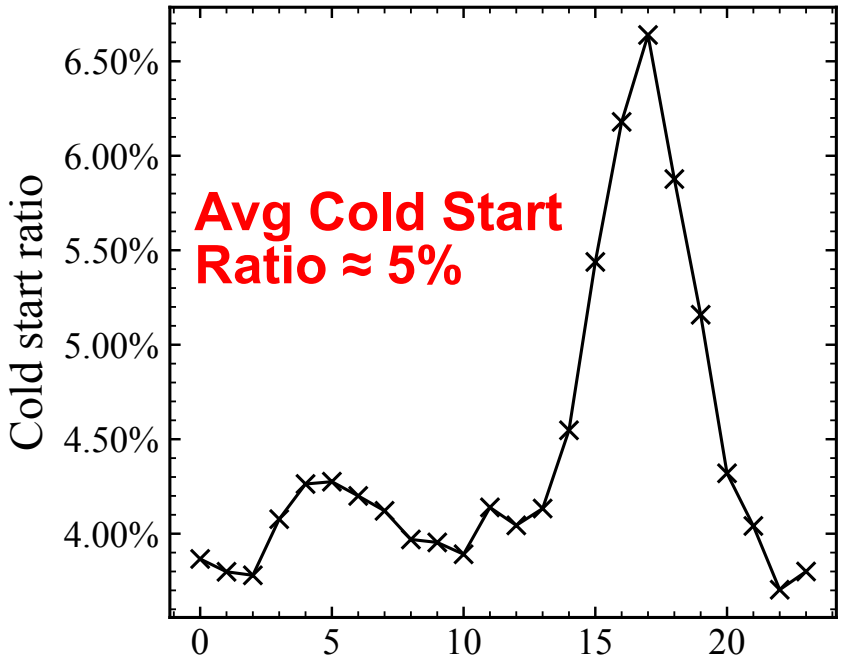
Alibaba Cloud

ANT GROUP

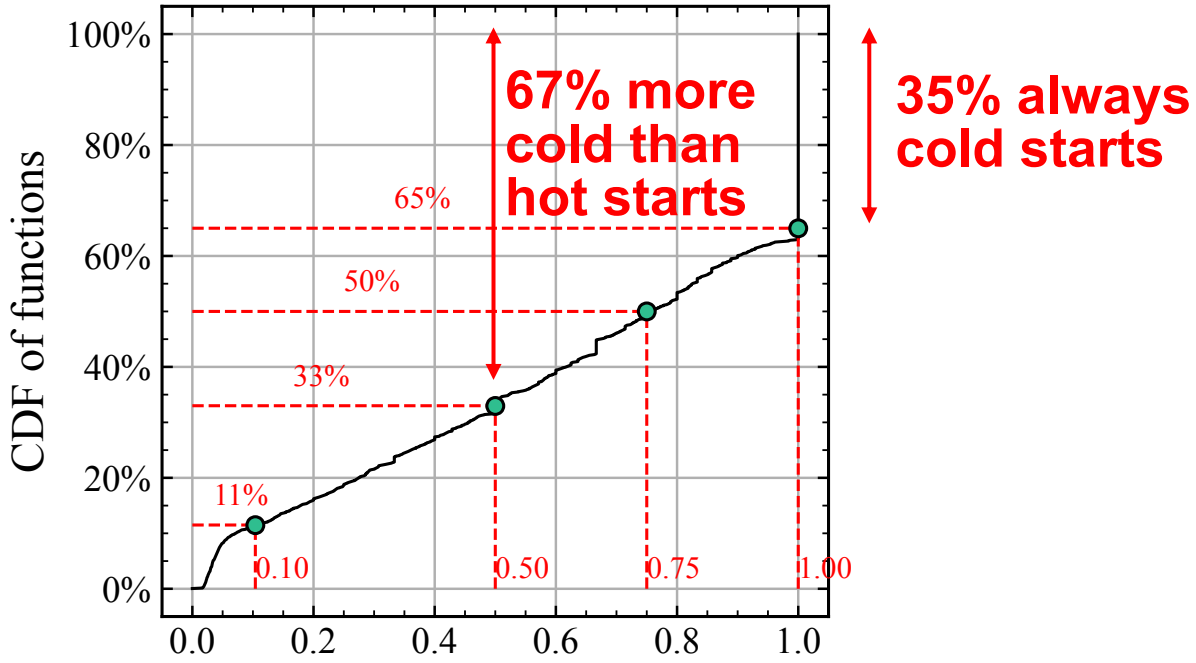
Google Cloud

Cold Start Problem Remains Critical

- **Ant Group Public Cloud – Invocation Characteristics**
  - 95% Hot starts, cover a **small set of high-frequency** functions (provisioned concurrency + 1min keep-alive)
  - Cold starts represent only 5% of invocations, yet they from the vast **majority of functions** across the platform



(a) **Cold Start Ratio during 24-hour Period**

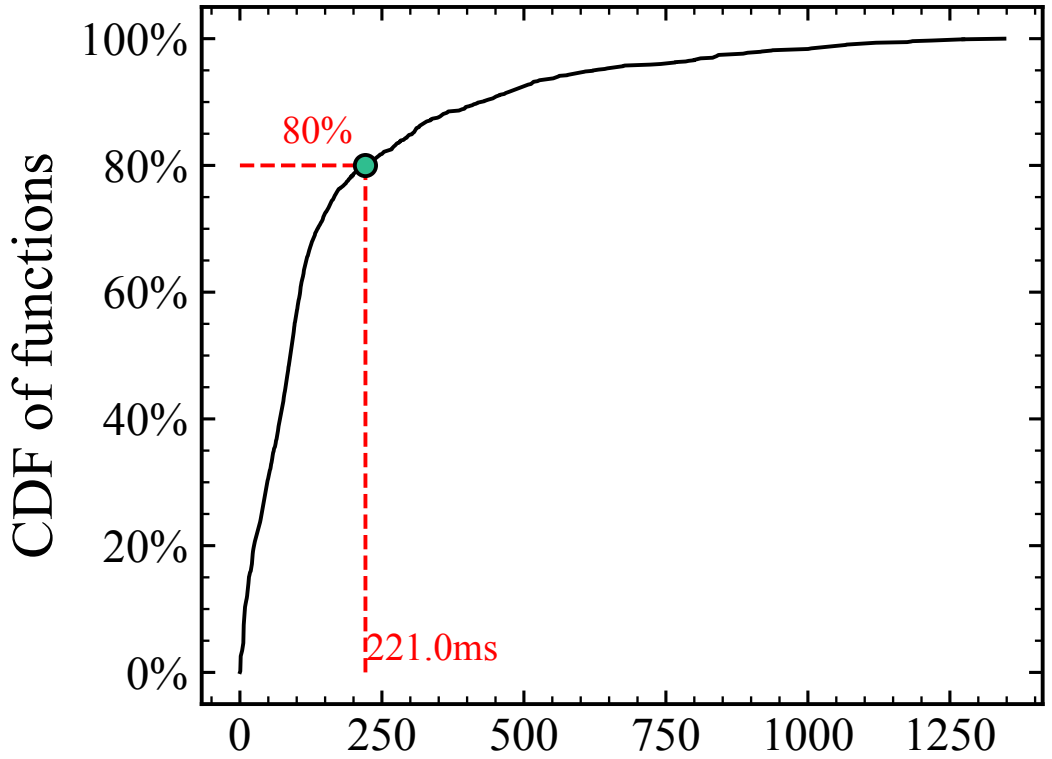


(b) **CDF of per Function Cold Start Probability**

Cold Start Problem Remains Critical

Ant Group Public Cloud: Execution Characteristics

- **80%** functions processed within **221ms**

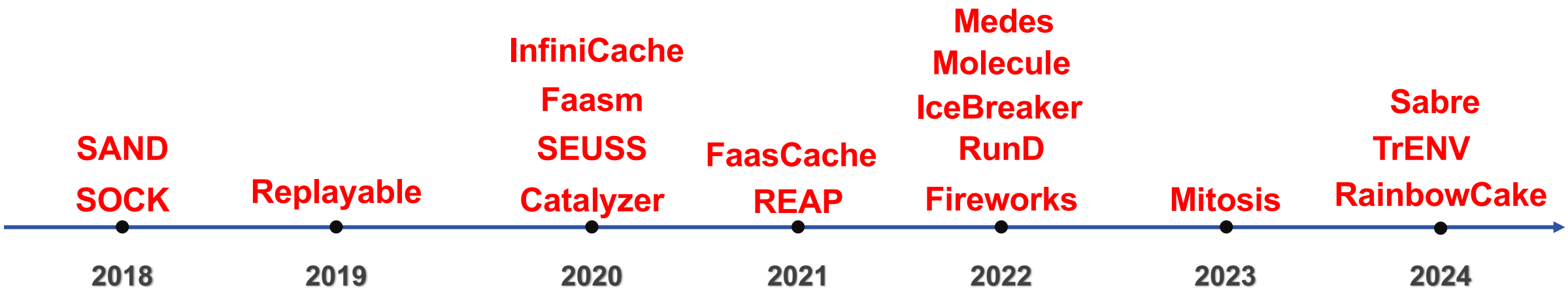


Short lifespans functions require (sub) millisecond cold start performance.

Serverless Research has Made Great Progress

Optimize cold starts: from 2018 to 2024

- Research papers* from top conferences have optimized instance initialization latency from hundreds of milliseconds to sub-millisecond



*: Disclaimer, we only select part of highly related works here and may omit some works.

How Far did We Go?

Current Serverless Research

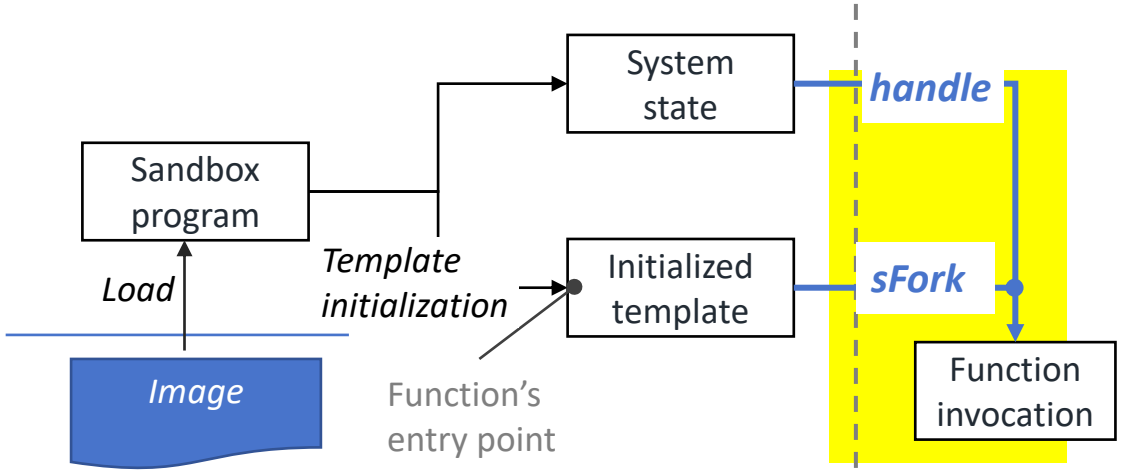


Still a large gap

Efficient, practical serverless systems

Practices of State-of-the-Art in Ant Group

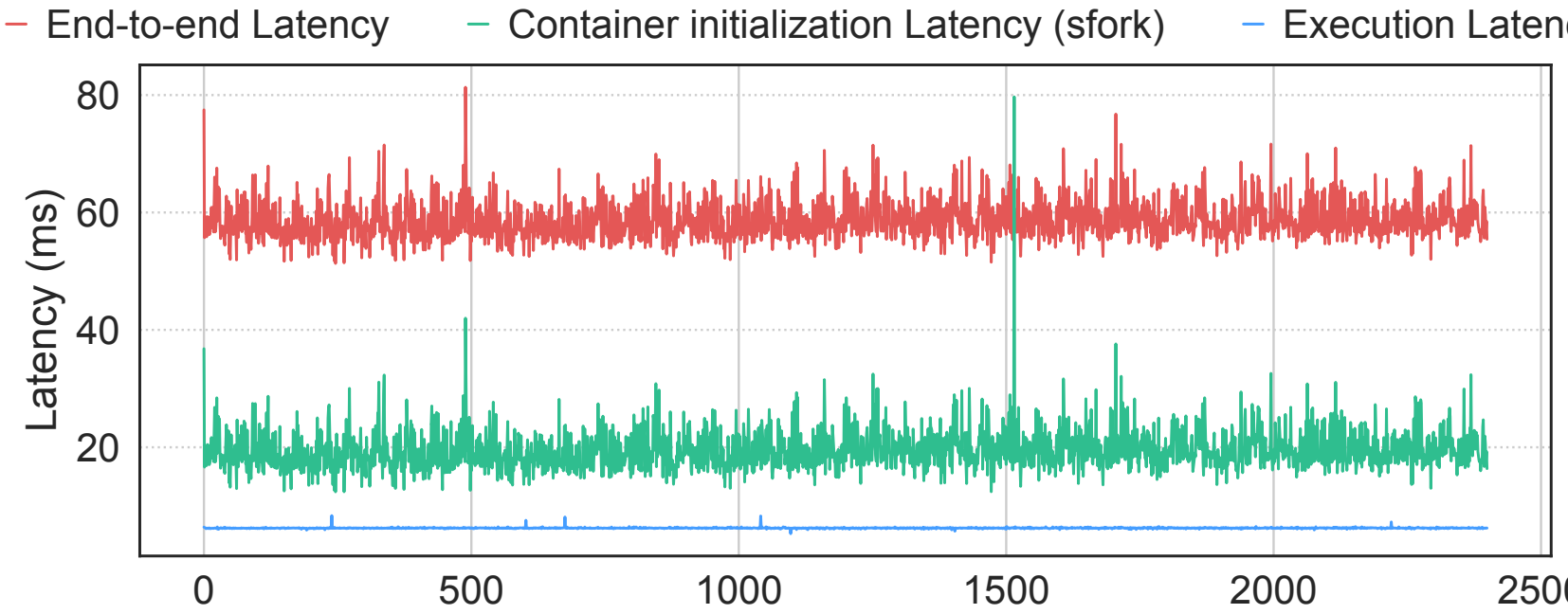
- **We have deployed state-of-the-art serverless system, Catalyzer[ASPLOS'20], in Ant Group**
- **Why Catalyzer ?**
  - 👍Secure container (based on gVisor)
  - 👍**Sub-millisecond startup** with sfork (sandbox fork)
  - 👍No extra hardware requirement
  - 👍Good compatibility (multi-process, unmodified applications…)
  - 👍Many others …
- However, although Catalyzer can achieve great container startup latency, it is **still insufficient** for optimal performance in real-world



***sfork* in Catalyzer**

Observation-1: Insufficient for End-to-End Latency

Container Initialization Latency + Execution Latency + ?? = Eend-to-End Latency



Avg E2E Latency ≈ 58.6ms

32.8ms gap

Avg sfork Latency ≈ 19.5ms

+

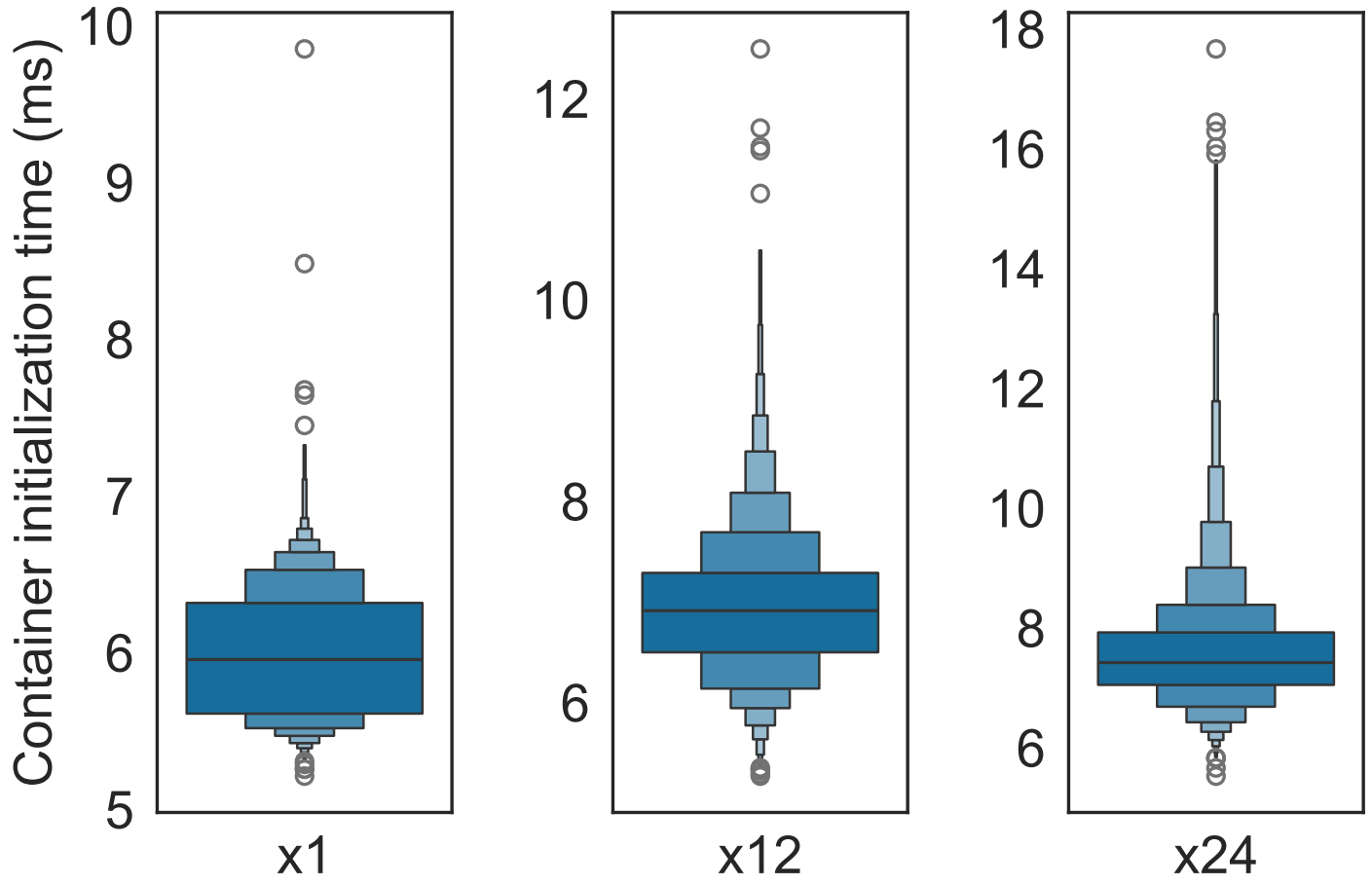
Avg Exec Latency ≈ 6.24ms

Sequential execution of a python-based function (JS, JSON Serde from function-bench[1])

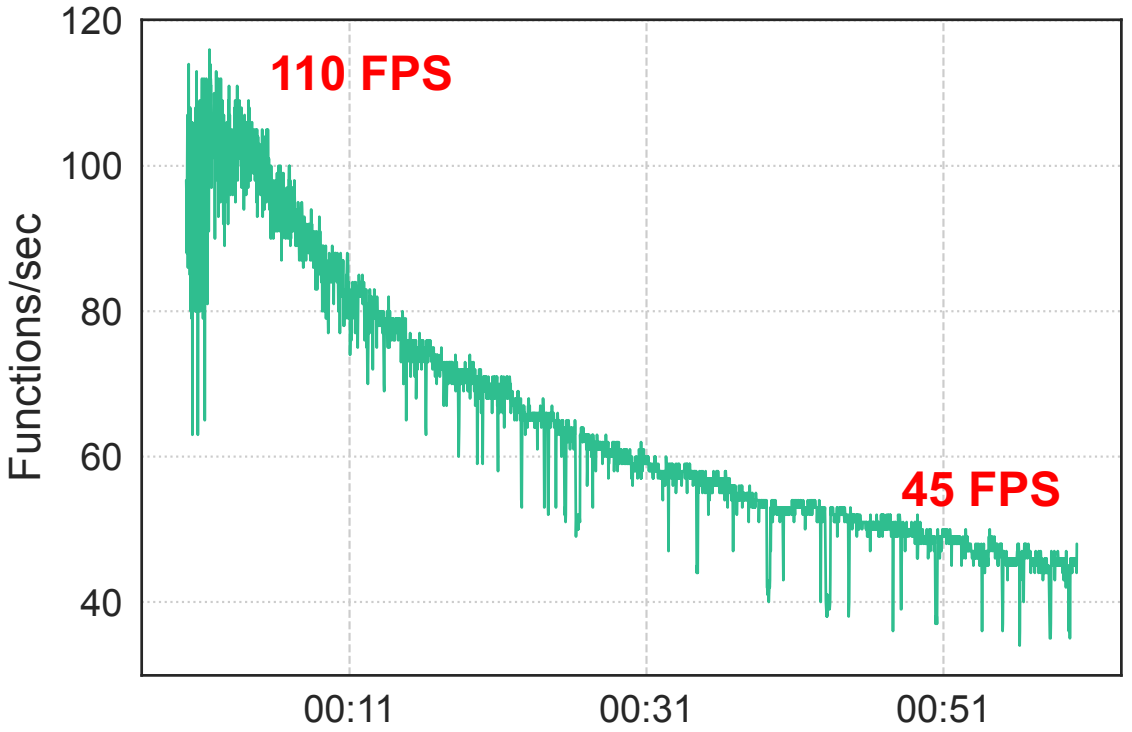
[1] Practical cloud workloads for serverless faas, SoCC'19

Observation-2: High tail latency under concurrent

[5.21ms, 9.85ms] **[5.27ms,12.49ms]** **[5.53ms,17.67ms]**

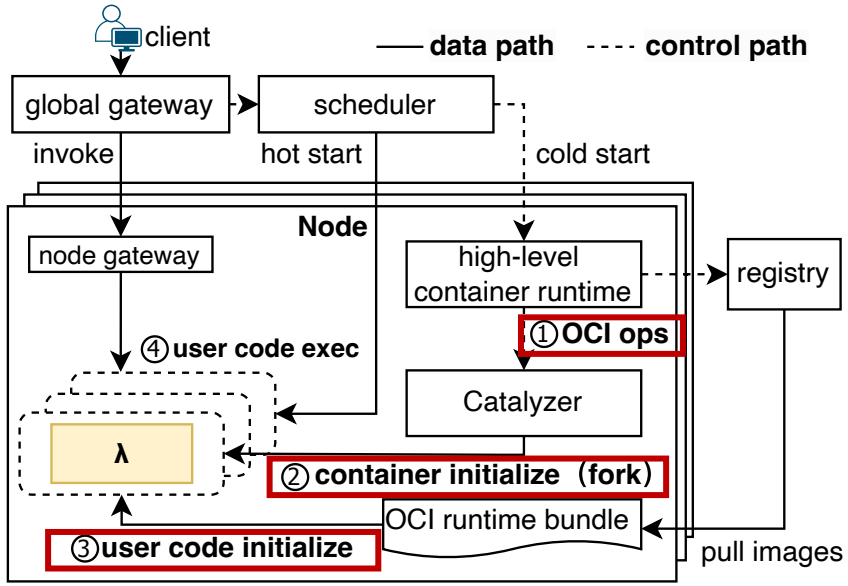


Observation-3: Performance degrades in sustained execution



Throughout of JS under x24 concurrent requests last for a full hour

Gaps from Industry Perspective for Fork-based Cold Start

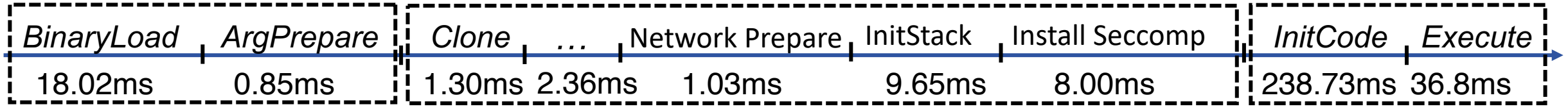


Fork-based serverless system (Catalyzer)

We identify three gaps:

- **Gap-1: Control-path latency (18–20ms)**
- **Gap-2: Resource contention latency (unstable)**
- **Gap-3: User code initialization latency (10ms–10s)**

Neglected **Heavily Optimized but Unstable at Scale** **Efficiency vs. Cost**

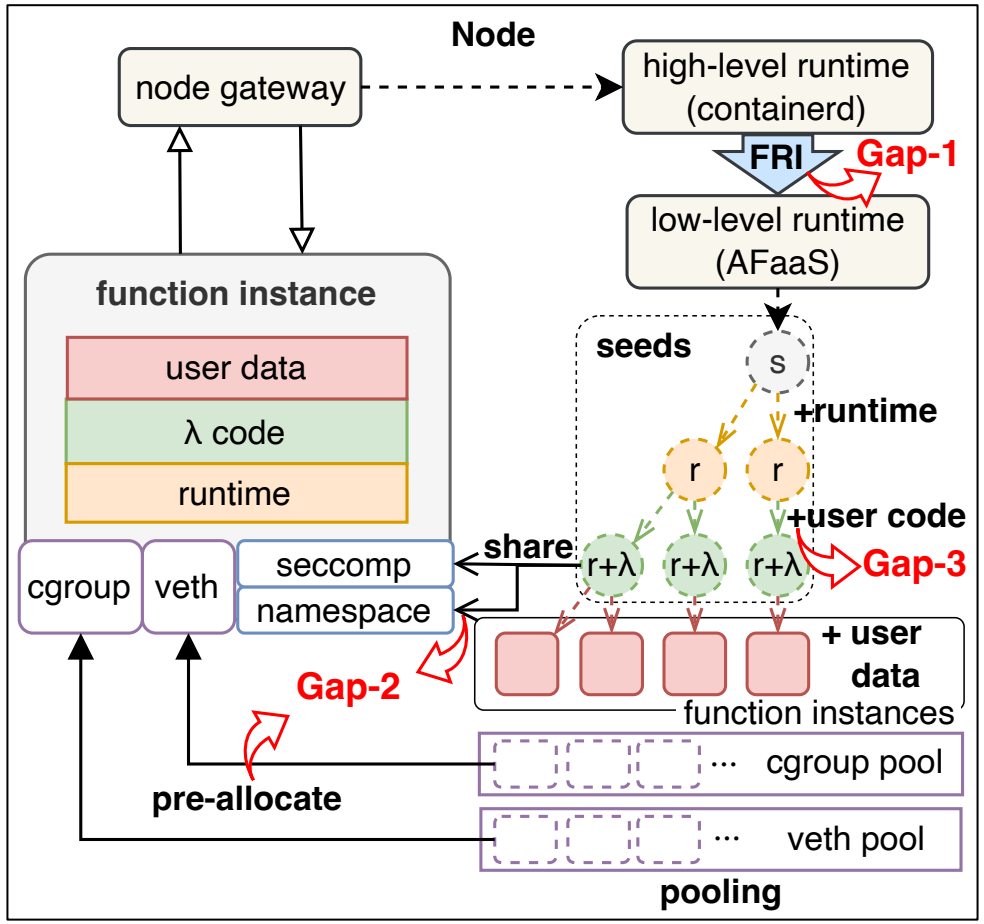


Breakdown of a production Node.js-based function

**OCI (Open Container Initiative) is an industry cloud-native standards container specification*

Our Solution: Ant FaaS (AFaaS)

Design-2: **Resource Pooling and Sharing** to alleviate resource contention



AFaaS Overview

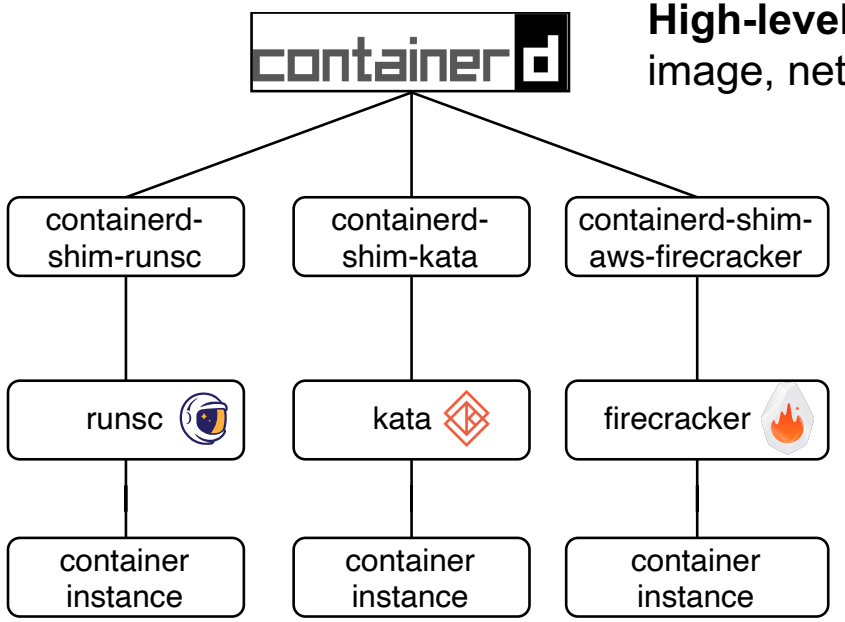
Design-1: Propose FRI to **Shorten Control Path**

Design-3: **Seeding User Code** to reduce user code load and initialization

Gap1: Control-path latency

OCI (Open Container Initiative)

- High modularity of the infrastructure
- Offer extensibility, but not efficiency for performance critical scenarios such as serverless

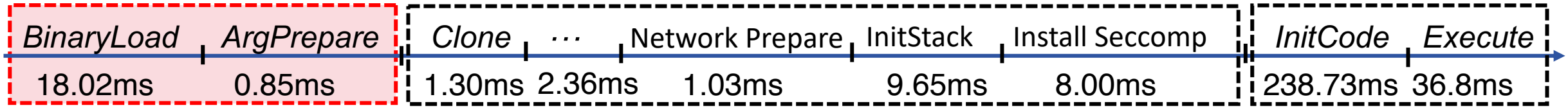


High-level runtime: lifecycle, image, networks management,

Shim: abstract low-level runtimes

Low-level runtime: command-line tool for spawning and running containers

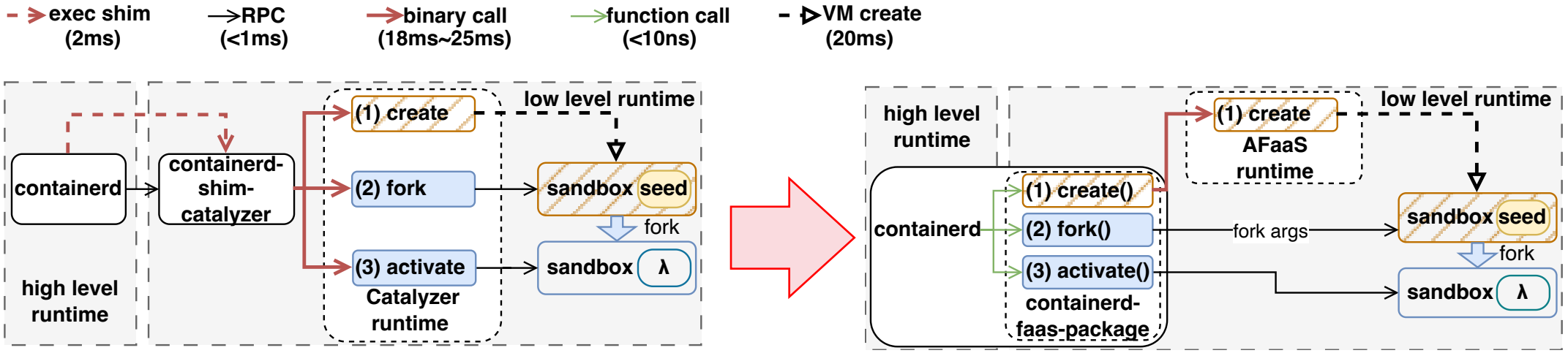
Neglected | **Heavily Optimized but Unstable at Scale** | **Efficiency vs. Cost**



AFaaS's Design#1-Shorten Control Path

FRI (Function Runtime Interface)

- a containerd-faas-package plugin (shim-less)
- a set Faas-oriention API (e.g, create, fork, and activate) that enable direct call from containerd to AFaaS (binary call-less)



OCI Specification (long path, heavy binary calls)

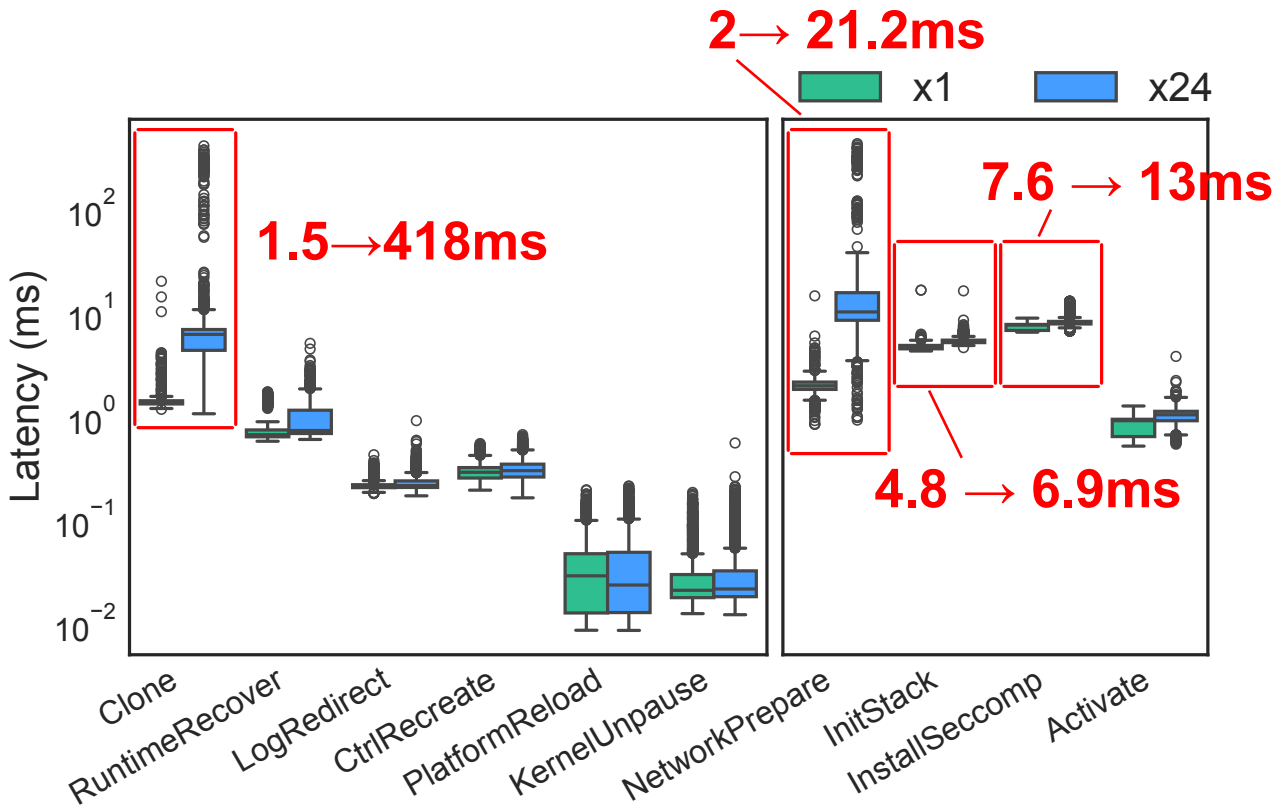
FRI Specification(shortened path, lightweight RPC calls)

Decision: *FRI demonstrates that we can sacrifice little modularity to unlock significant efficiency gains in cold starts.*

Gap-2: Resource contention latency

Kernel Resource or Operation

- **Clone**: IPC and Network namespace (sb_lock, mount_lock, superblock cache)
- **Network Preparation**: parse config.json, *setns(), netlink()*
- **Network Stack**: complex data structures (e.g., ring buffers)
- **Seccomp**: BPF JIT
- **Others**: cgroup



Tail latency caused by kernel resource contention

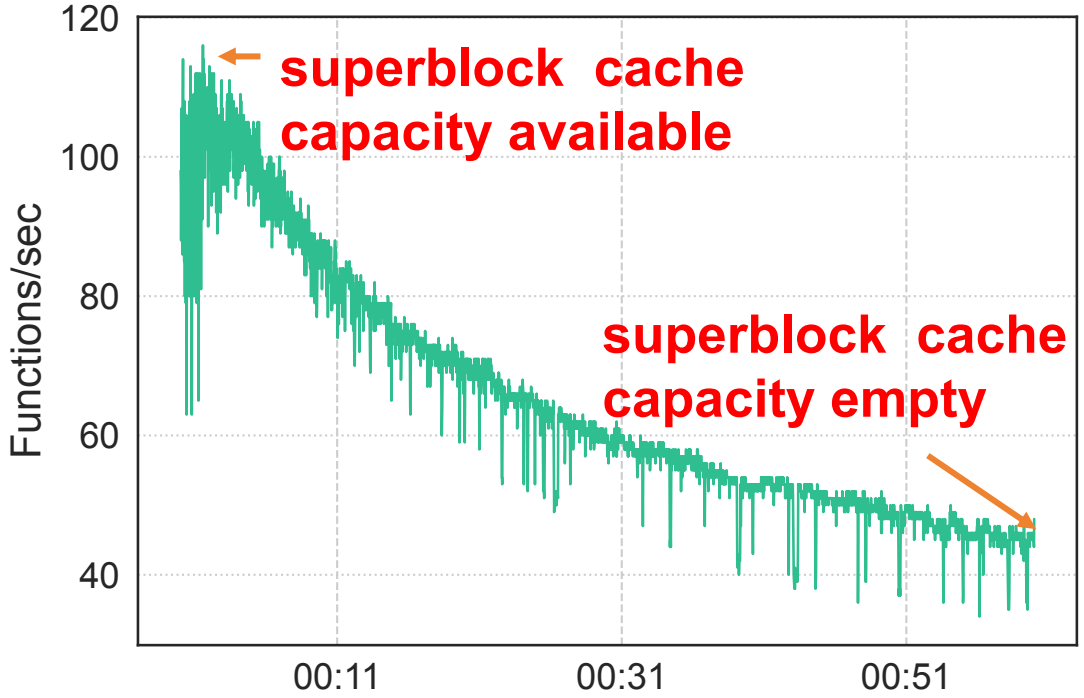
Neglected | **Heavily Optimized but Unstable at Scale** | **Efficiency vs. Cost**

BinaryLoad	*ArgPrepare*	*Clone*	⋯	Network Prepare	InitStack	Install Seccomp	*InitCode*	*Execute*
18.02ms	0.85ms	1.30ms	2.36ms	1.03ms	9.65ms	8.00ms	238.73ms	36.8ms

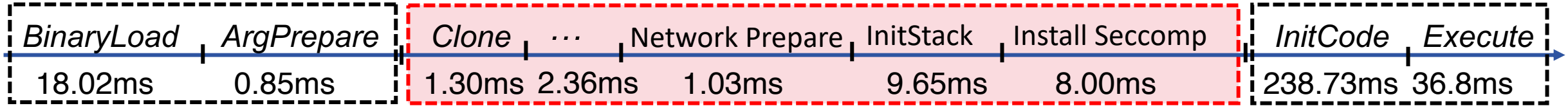
Gap-2: Resource contention latency

Kernel Resource or Operation

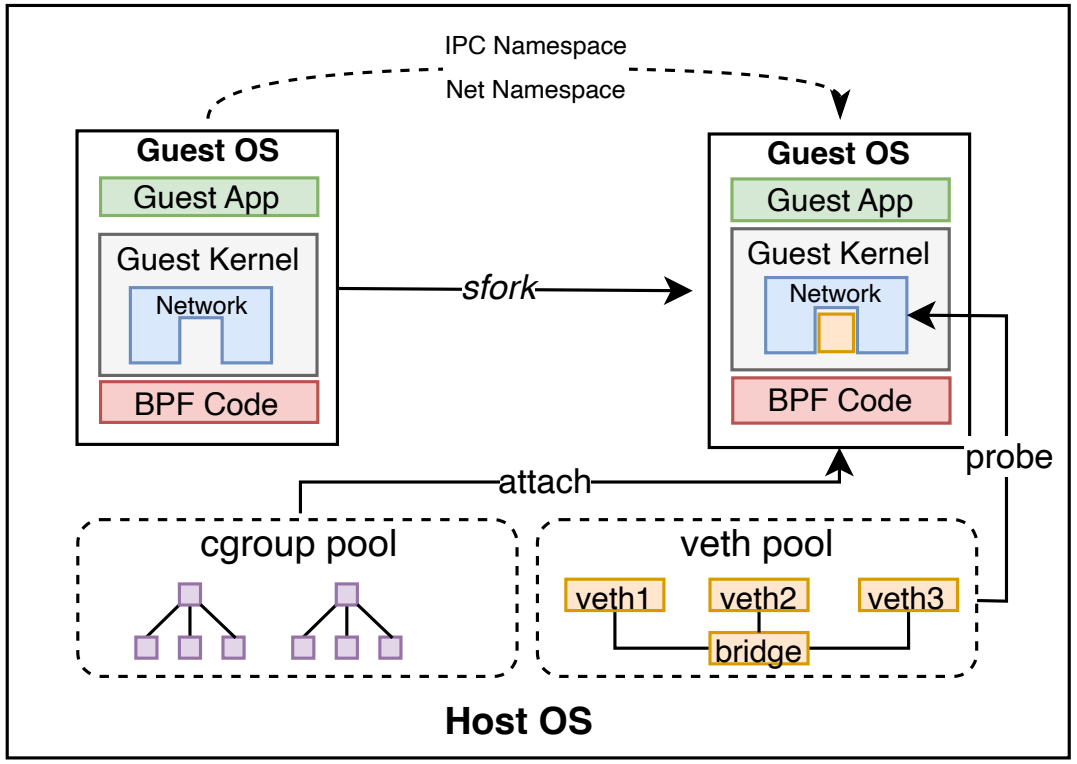
- **Clone**: IPC and Network namespace (sb_lock, mount_lock, superblock cache)
- **Network Preparation**: parse config.json, *setns(), netlink()*
- **Network Stack**: complex data structures (e.g., ring buffers)
- **Seccomp**: BPF JIT
- **Others**: cgroup



Neglected | **Heavily Optimized but Unstable at Scale** | **Efficiency vs. Cost**



AFaaS's Design#2-Resource Pooling and Sharing



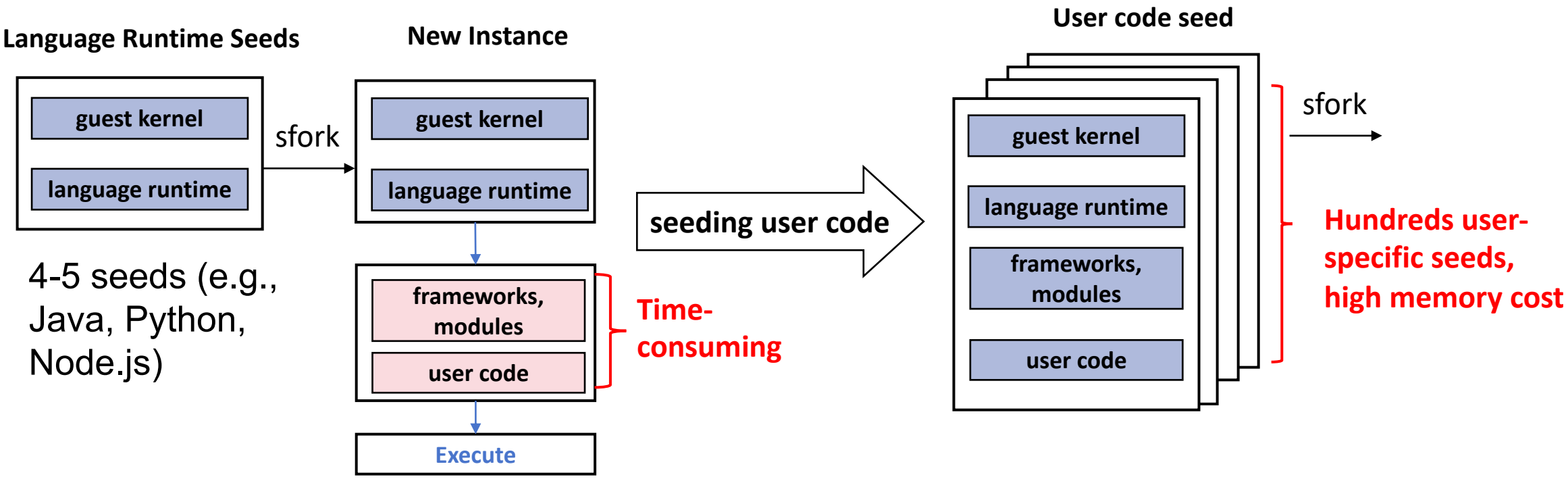
Combines:

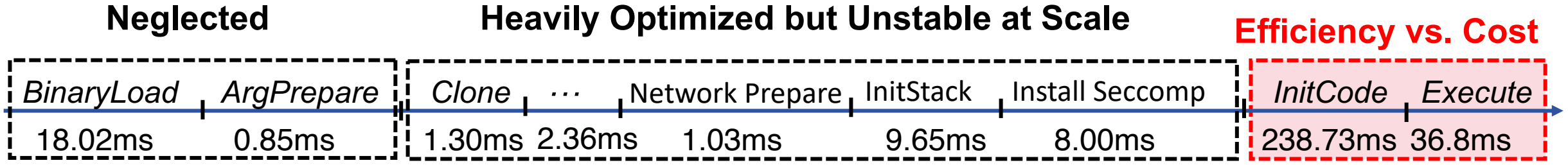
- **Pooling:** cgroup, veth
- **Sharing:** namespace, network stack, seccomp

to alleviate resource contention

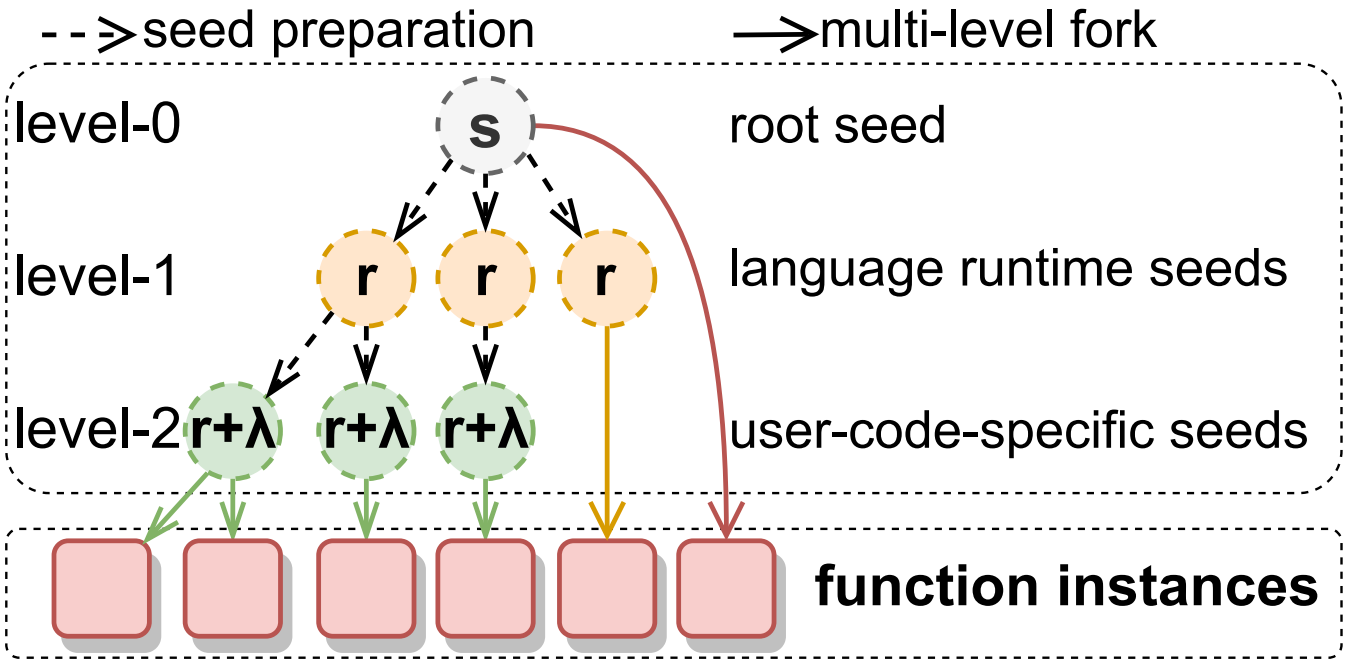
Decision: In high-density scenarios, pooling/sharing high-contention elements are effective strategy to retain the sandbox fork's low-latency performance

Gap-3: User code initialization latency





AFaaS's Design#3 - Seeding User Code



Combines:

- **Tree-structured Seeds** to maximum memory sharing across levels and nodes
- **Multi-Level Fork** from the **nearest viable ancestor** to gain deterministic millisecond cold starts

Decision: Tree-structured seeds with multi-level forking enable adaptive cold starts, enabling maximum state reuse and best-effort performance.

Evaluation

Baseline:

CataOnly	Enhanced Catalyzer
CataOPT1	***CataOPT1*** = ***CataOnly*** + ***Gap2-OPT***
CataOPT2	***CataOPT2*** = ***CataOPT1*** + ***Gap1-OPT***
AFaaS	***AFaaS*** = ***CataOPT2*** + ***Gap3-OPT***
Kata	Initializes by cloning from a pre-created VM (Kata-template), v3.15.0.
gVisor	Starts from scratch, release-20250414.0.

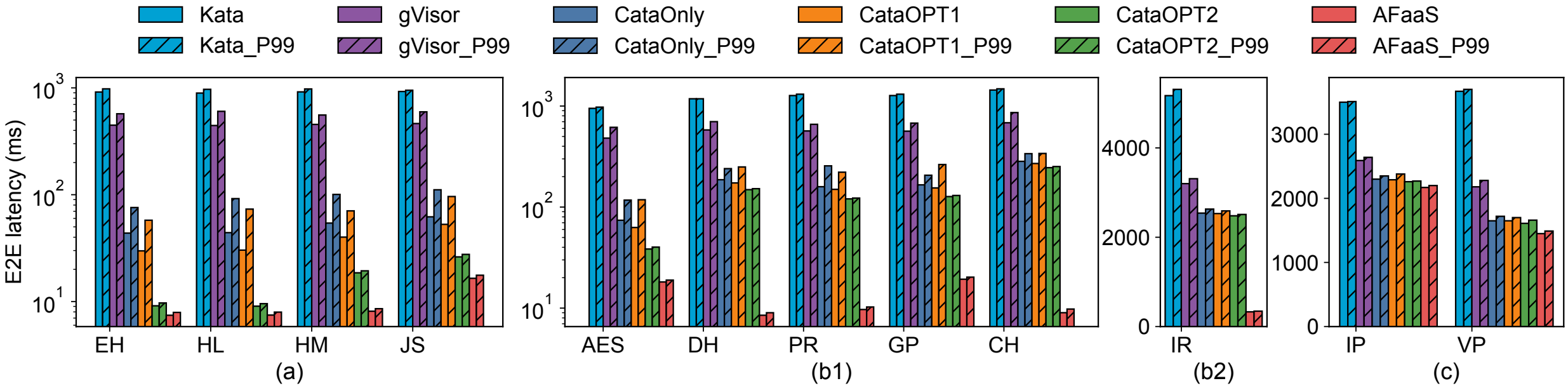
Experimental Setup:

- 24 cores Intel(R) Xeon(R) Platinum 8163 CPU, NUMA off
- 2.50GHz
- 512GB RAM
- Linux 5.10

Functions Evaluated:

- 12 functions from Function-Bench
- 8 functions from production

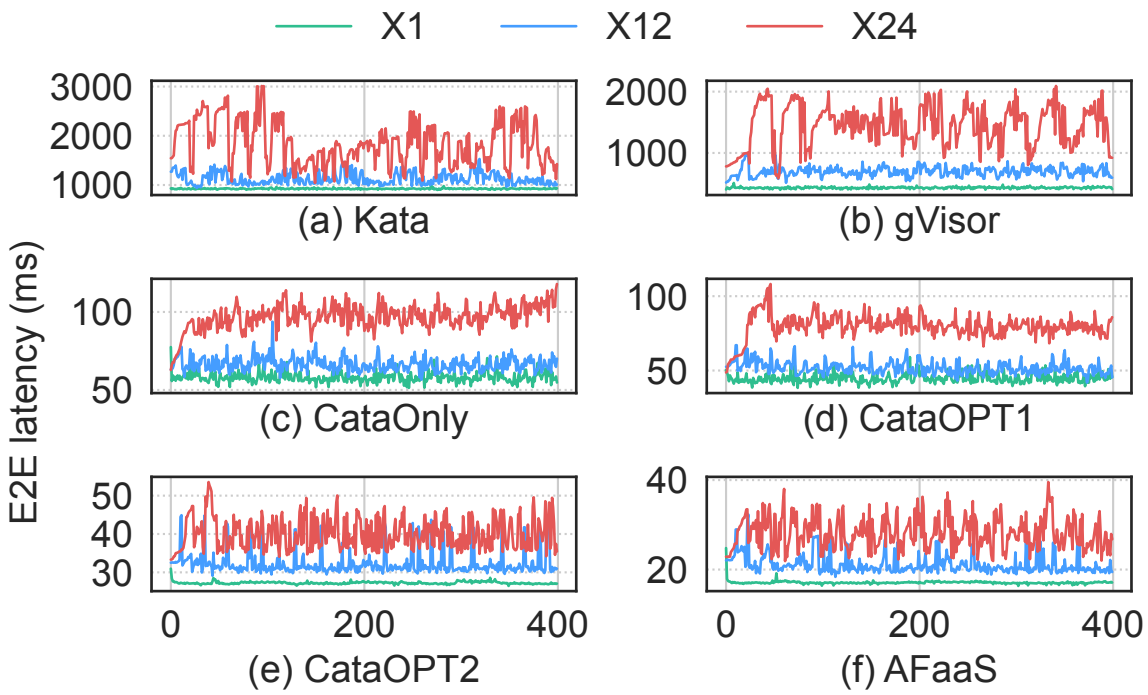
End-to-End Latency (sequential)



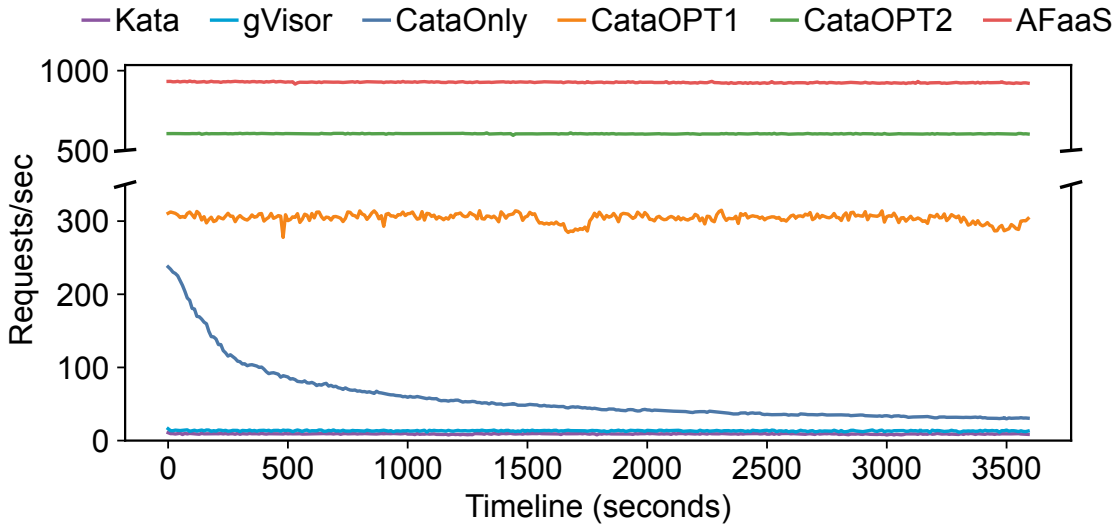
		User code initialization time	
		Short	Long
User code exec time	**Short**	**3.76x–6.68x (a)**	**4.09x–31.48x (b1/b2)**
	Long	1.05x-1.14x (c)	

Most functions in Ant Group are short

Concurrency and Sustained Traffic Performance



E2E latency under different levels of concurrency



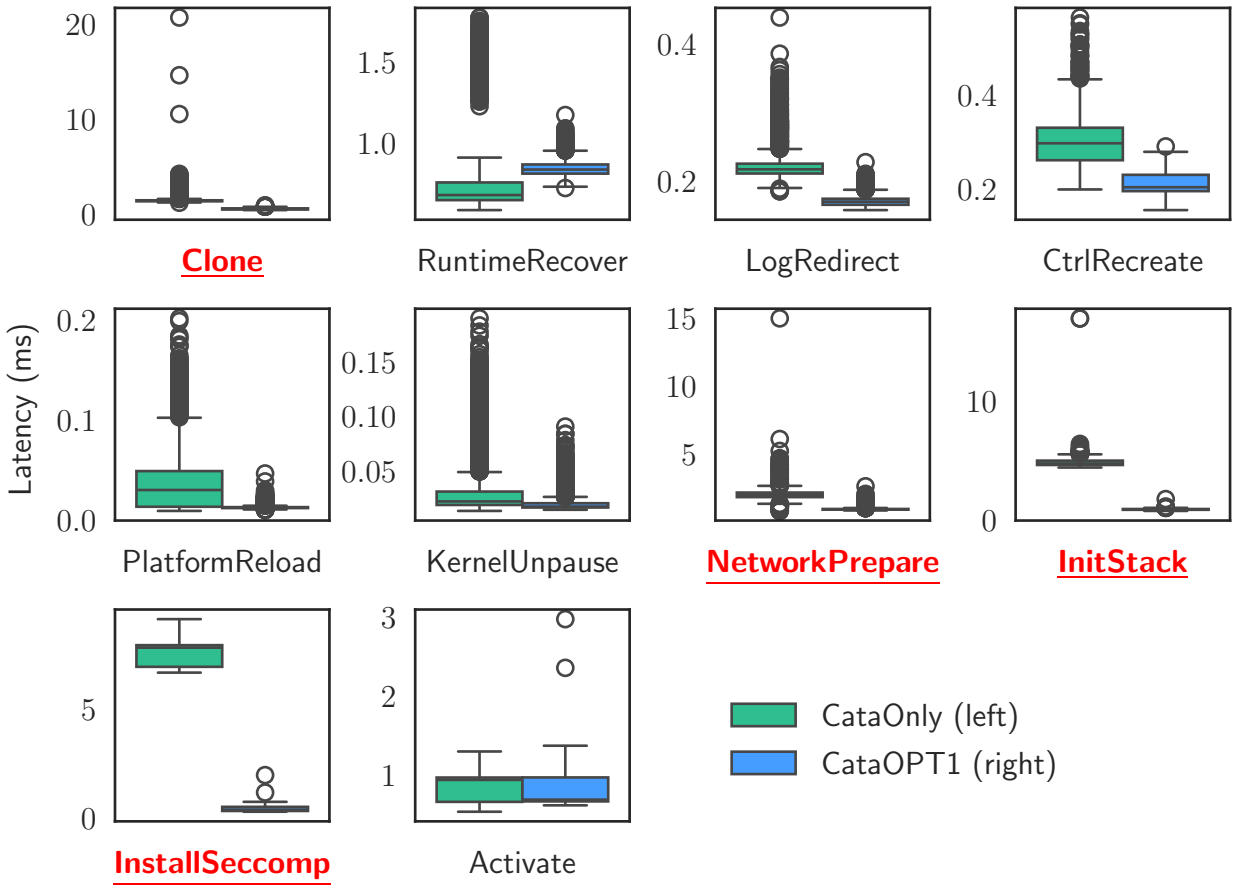
Throughput variation over time at high sustained concurrency (x24)

Smallest tail latency under all concurrency

AFaaS	[6.97ms, 14.55ms]
CataOnly	[38.39ms, 74.05ms]

Maintain very stable throughput

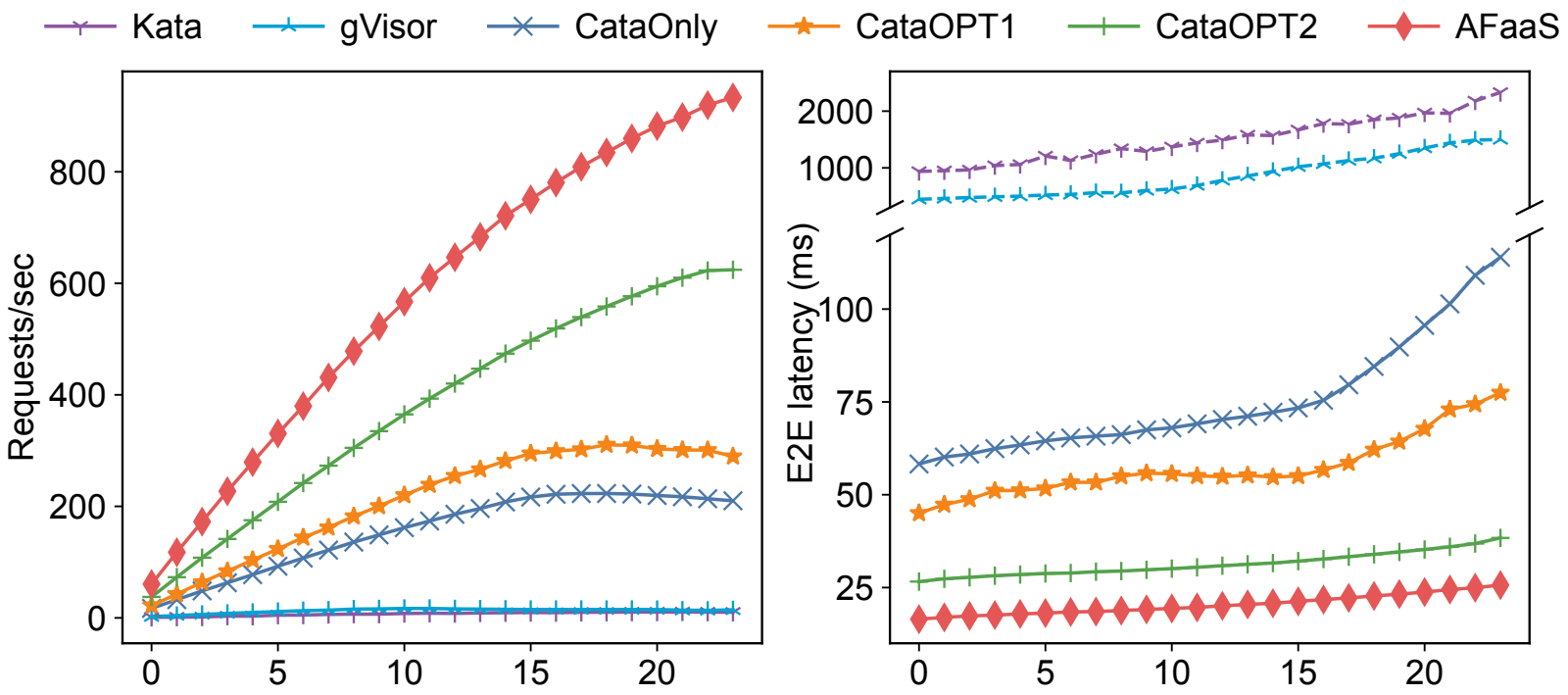
Optimization Breakdown



The tail latency of **optimized phases** and **unoptimized phases** is both reduced

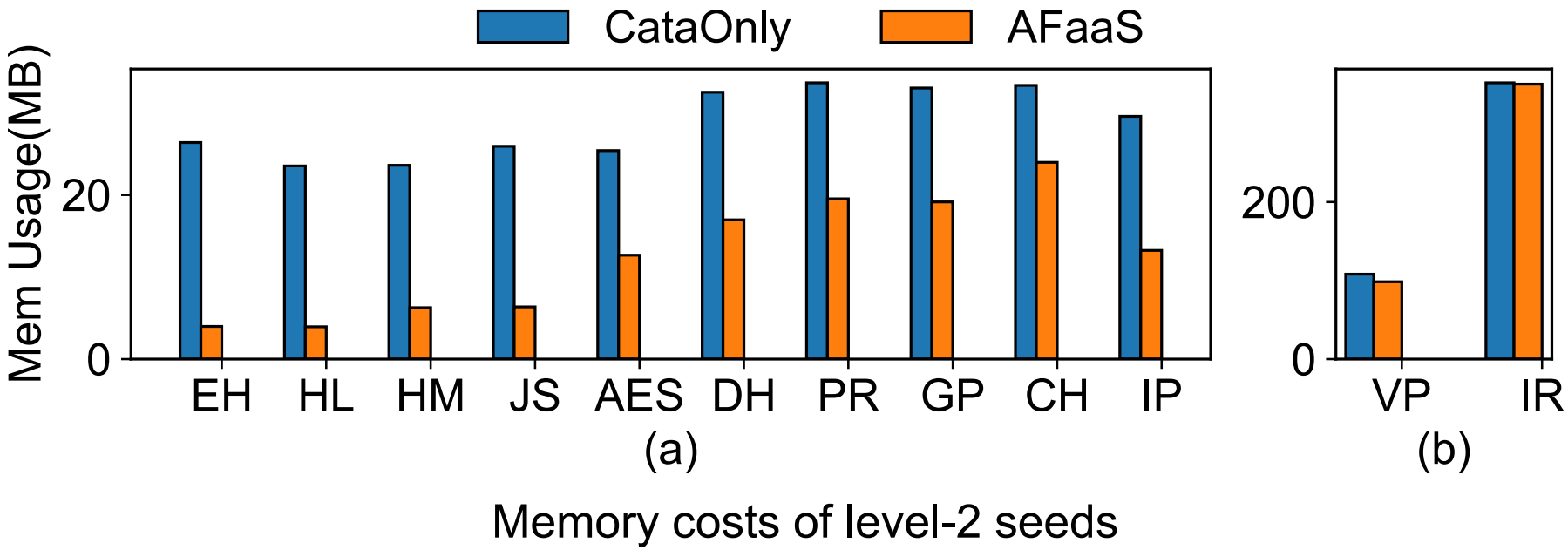
Latency fluctuation of different phases during container initialization (×24)

Scalability



Throughput (left) and E2E latency (right) with varying levels of concurrency

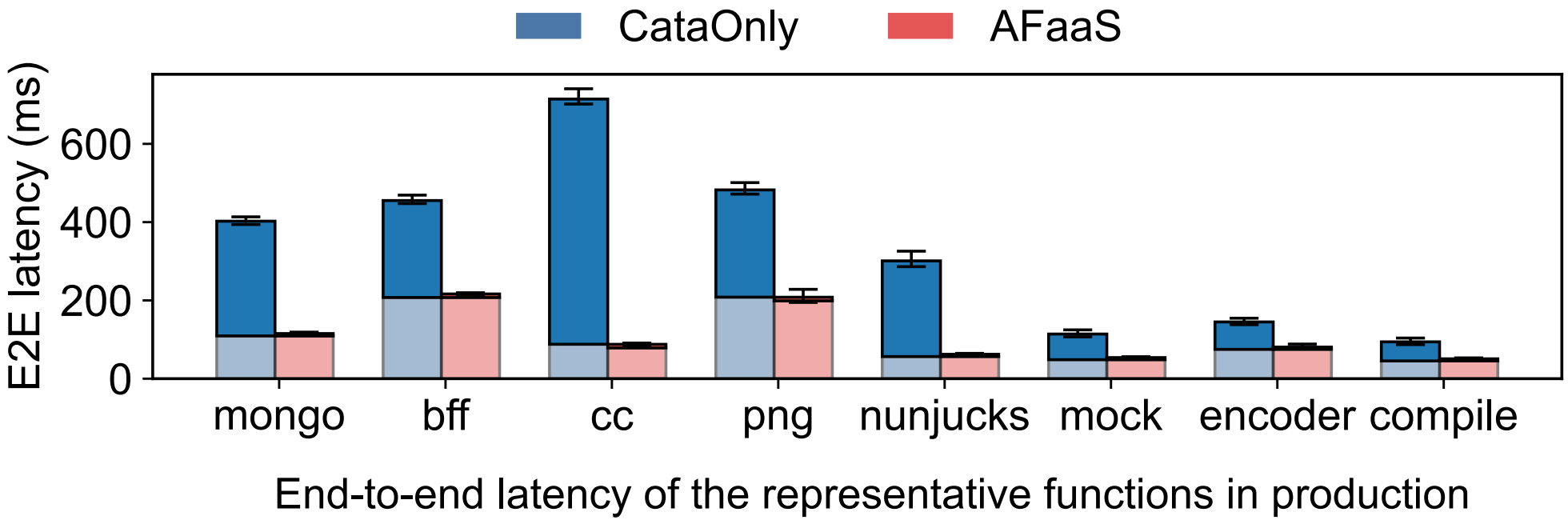
Memory Cost



Memory usage reduced

User code size	
Small	Large
28.1%–84.9%	less than 1%-9%

Real-World Workload in Production



End-to-end latency of the representative functions in production

End-to-end performance: 1.8x to 8.1x compared to CataOnly
Cold start time: [5.5ms, 9.4ms]

Adoption of AFaaS

Public Cloud

- **Serverless Platform:** 50,000+ unique FaaS functions deployed, 100M+ daily executions, P99.99 latency < 20ms (incl. central scheduler overhead)

Private Cloud

- **Microservices:** 5,000+ microservices, 140M+ daily RPC invocations
- **AI Workloads:** Trained on 300B+ data records and 100M+ code interpreters
- **Additional Workloads**: Big-data analytics, Test/CI frameworks…

Lessons and Experiences

Fork's Renaissance in Serverless Systems

- Caching: unpredictable invocations, high resources costs
- C/R: I/O-bound
- Fork: great potential for building a high-performance, stable, and efficient serverless system based on the design presented in this paper.

AFaaS Deployment Experience

- Securely handle uniqueness: re-initialization random numbers/cryptographic tokens
- Prepare resource pools sequentially: avoid cb_mutex contention in parallel netlink() ops
- Bottlenecks under ultra-high concurrency: seed, co-located workloads
- See the paper for further discussion

Conclusion: Fork in The Road

AFaaS: From fork prototype to production

- Identify three key bottlenecks: (1) **inefficient control-path interactions**, (2) **resource contention under concurrency**, and (3) **user code initialization overhead**
- Propose three practical design: (1) **incorporates FRI to shorten control path**,(2) **pools and shares resources to alleviate contention**, and (3) uses **tree-structured seeds** for fast and efficient fork

Open-source trace: https://github.com/antgroup/AFaaS
Service online: https://cloud.alipay.com/main/product/cloudbase

Thank you!