ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

USENIX ATC 2022

RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing

Zijun Li, Jiagan Cheng, and Quan Chen, Shanghai Jiao Tong University;
Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, and Weidong Han, Alibaba Group;
Minyi Guo, Shanghai Jiao Tong University

Emerging Parallel
EPCC
Computing Center

Alibaba Cloud

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Introduction & Background

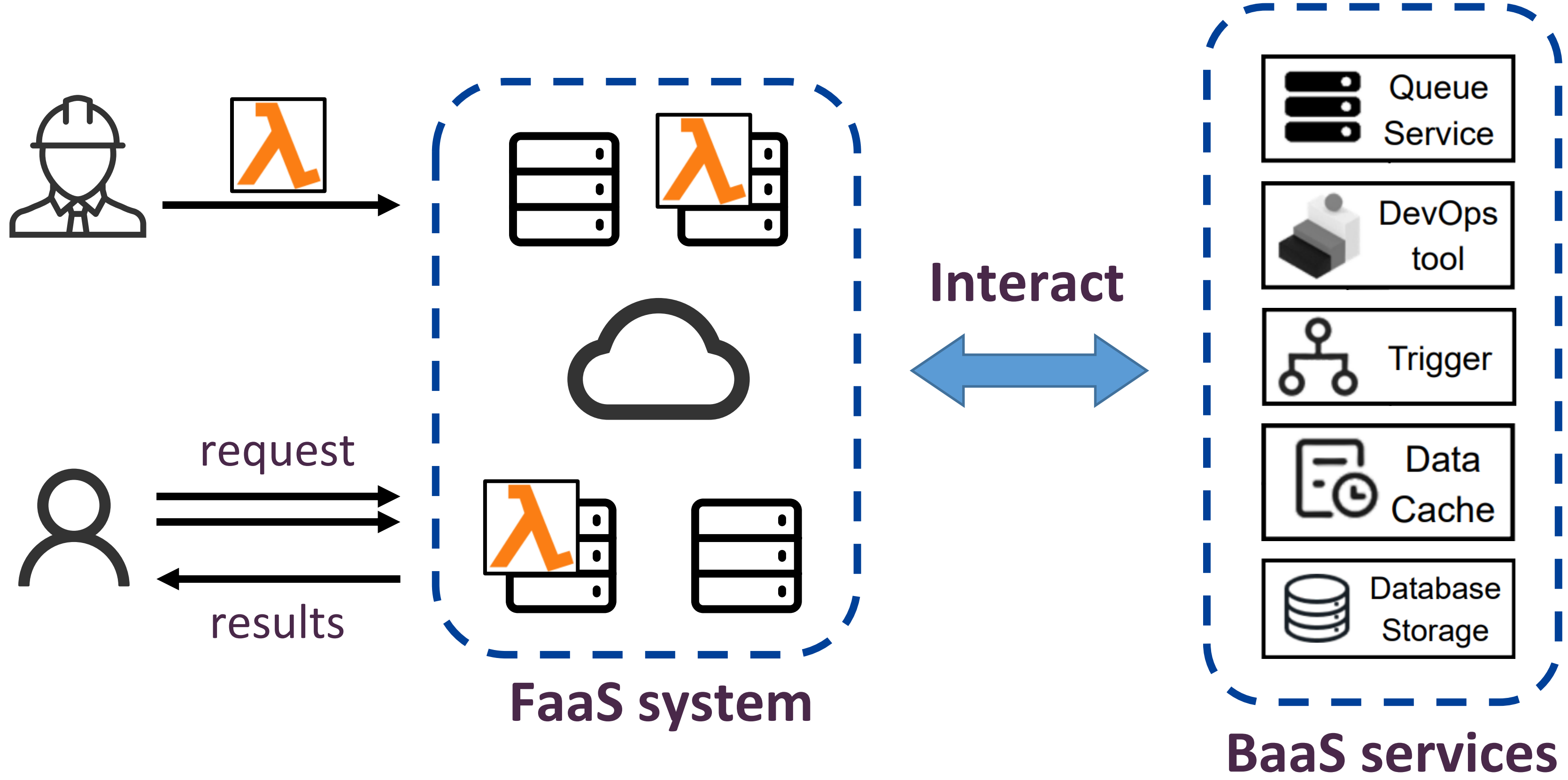
- Definition of serverless (FaaS).
- Challenges of multi-tenants in serverless.

RunD

Introduction & Background

What is Serverless?

*Berkerly's View: "**Serverless** = **FaaS** (Function-as-a-Service) + **BaaS** (Backend-as-a-Service)"*



Introduction & Background

How to guarantee the security with multi-tenants?

- ***Normal containers*** *(like runc, LXC).*
  - *Based on namespace, cgroups* — ✗ ***Weak isolation***
  - *Share Host kernel* — ✓ ***Low overhead***

- ***Secure containers*** *(like FireCracker, Kata Containers).*
  - *Hypervisor-based virtualization* — ✓ ***Strong isolation***
  - *Need to load guest kernel* — ✗ ***Low overhead***

Introduction & Background

Characteristics in Serverless computing

- *Most functions with small container specification*

  E.g., 47% of lambda functions -> 128MB

- *Actual memory usage is much smaller*

  E.g., 90% of Azure applications < 400MB

- *Multiple function invocations may arrival in a short time*

  E.g., 200+ container-launch requests within 1s.

- *Thousands of containers*

  *E.g., a node with 256GB -> max 256*1024/128 = 2048 containers*

Basic guarantee

Low response latency

Two requirements

High-concurrency startup

High-density deployment

Introduction & Background

What's the limitation of using Secure Containers in Serverless?

- ***Observation in high-concurrency scenario (>100-way)***
  - *Distinct performance degradation of creating containers (10s)*
  - *High CPU time and scheduling overhead*
- ***Observation in high-density scenario (>1000 containers)***
  - *MicroVM components occupies most of memory space*
  - *Degradation of containers' runtime performance (1.5x slower)*

Current Secure Containers have concurrency and density bottlenecks!

Introduction & Background

Motivation

- What are the bottlenecks of serverless?
- Where do these bottlenecks come from?

Motivation

The *rootfs* mounting for density/concurrency requirements:

- **Virtio-blk** (based on block devices).

  good performance of rand/seq read/wirte.

  time-consuming of preparing LVs in high-concurrency

  double page cache in high-density

- **Virtio-fs** (based on filesystem sharing).

  good performance of rand/seq read except write

  enable sharing page cache

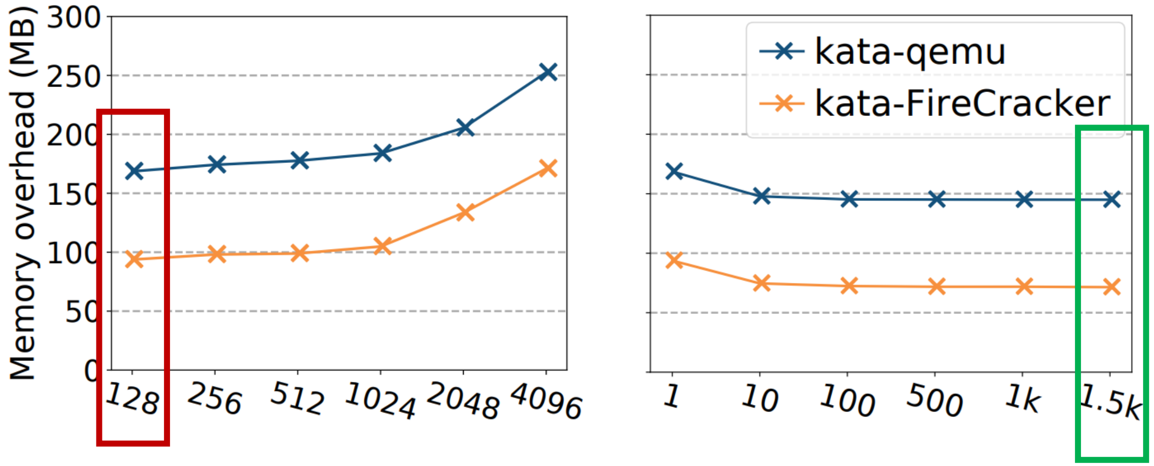
  daemon-per-container introducing high CPU overhead in high-density

The current secure container fails to discriminate between serverless platforms and traditional infrastructure-as-a-service environments.

Motivation

Memory footprint of MicroVM for density requirements.

- *GuestOS, struct page, shimv2, agent, …*



(a) Impact of specification (solo) (b) Impact of density (128MB)

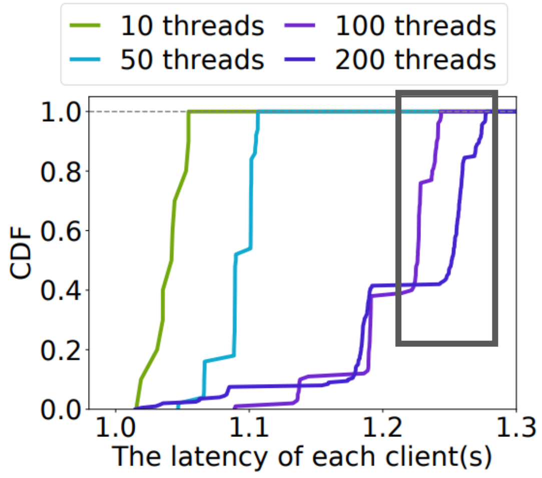
the memory overheads of a 128MB container are 94MB and 168MB with Kata-FireCracker and Kata-qemu

the per-microVM memory overhead reduce to 145MB and 71MB across 1000+ VMs. The overhead is still too large for a container with only 128MB memory specification

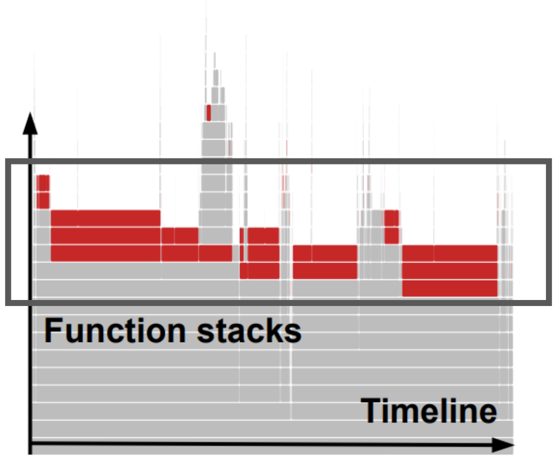
Motivation

Serialized cgroups operations for concurrency requirements.

- 100+ clients commit cgroups operations
- 1000+ cgroups operations per second
- 10000+ cgroups maintained in host



(a) Latency distribution



(b) The flame graph of Perf

(1) Mutex locks serialize the operations of cgroups.

(2) Spinner cgroups experience the optimistic spinning.

(3) Failure to acquire the lock will drag down tail latencies.

Introduction & Background

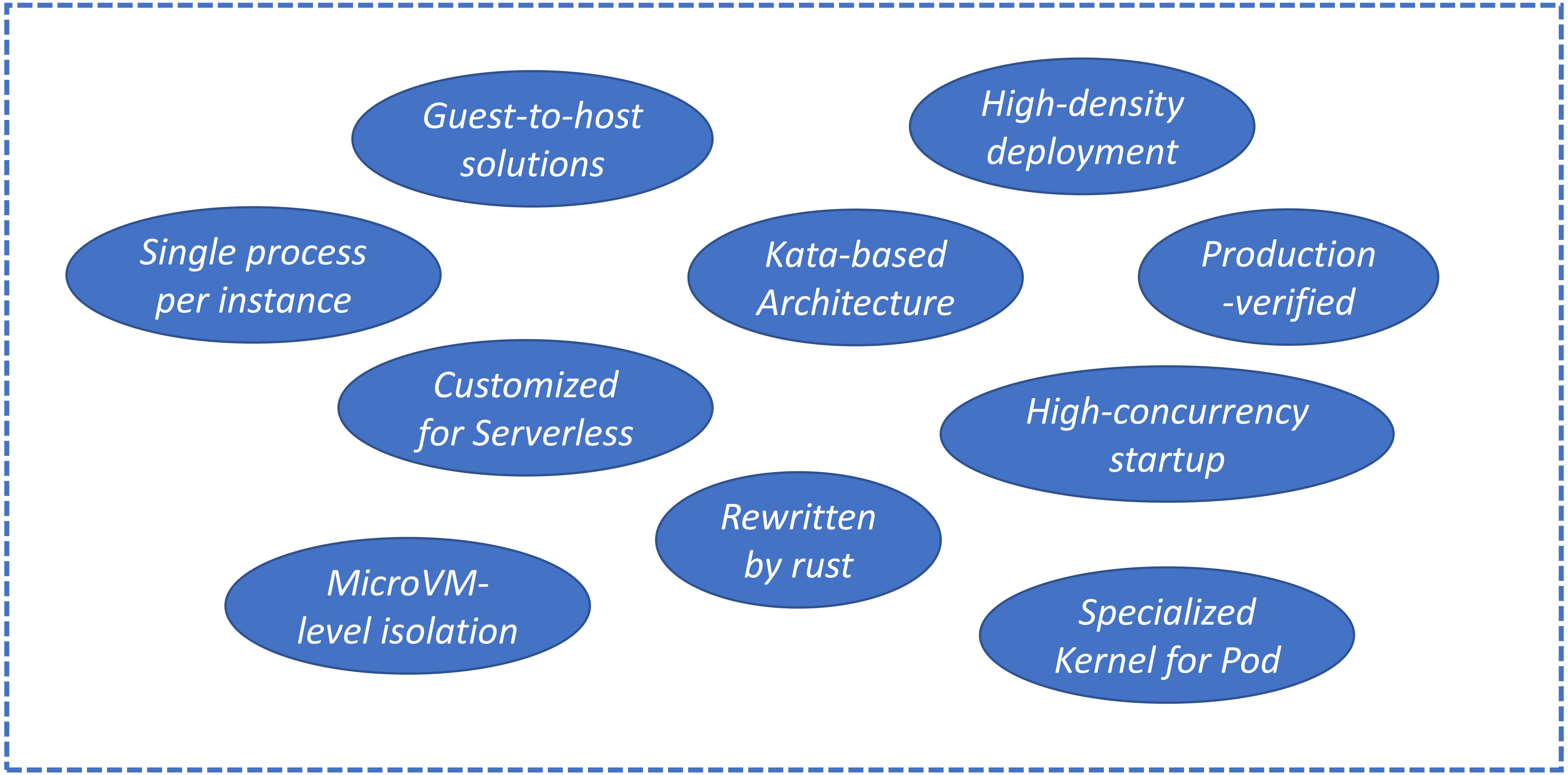
Motivation

Methodology & Design

- Lightweight Serverless Runtime - RunD
- Read-write splitting rootfs
- Condensed kernel and pre-pateched image
- Lightweight cgroups with cgroup pool

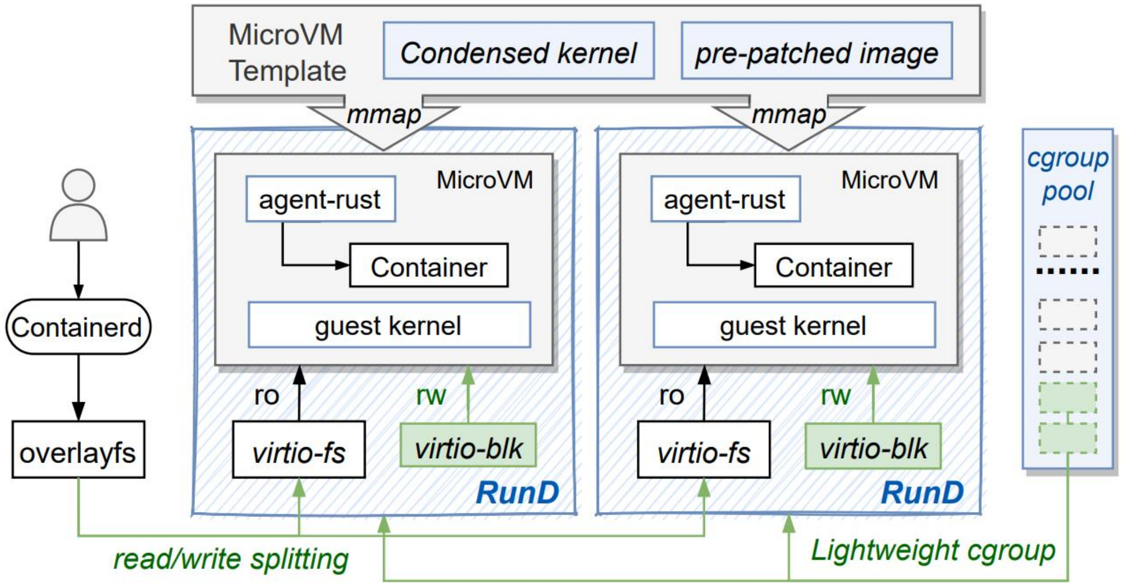
Methodology & Design

Lightweight serverless runtime - RunD



Methodology & Design

Lightweight serverless runtime - RunD



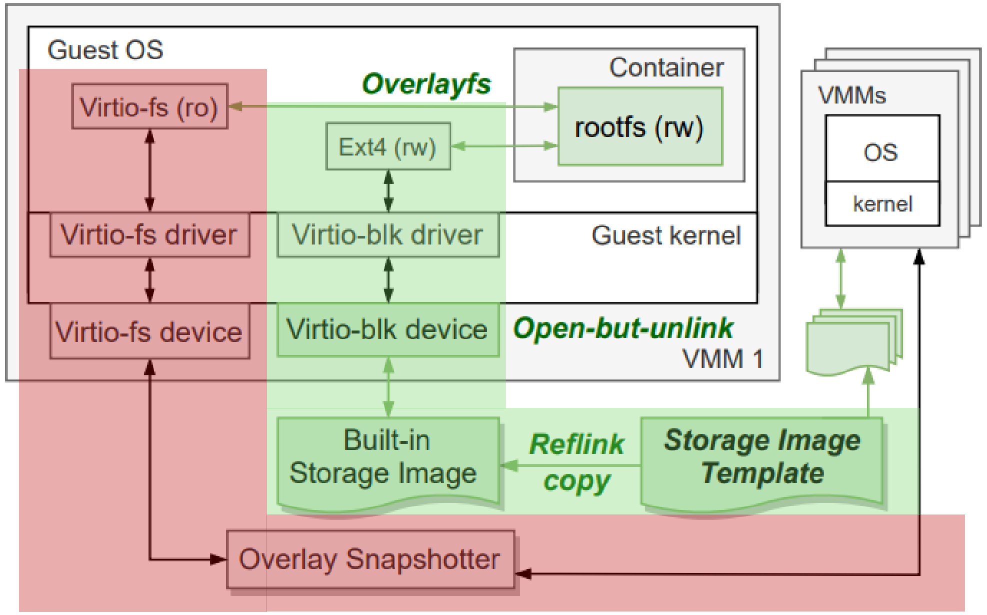
- *Step 1: containerd -> RunD runtime*
- *Step 2: runc-container rootfs (ro and rw) -> VMM.*
- *Step 3: MicroVM template -> sandbox.*
- *Step 4: lightweight cgroup -> attached to sandbox.*

Guest-to-Host optimizations

Methodology & Design

Efficient container rootfs mapping leveraging serverless features

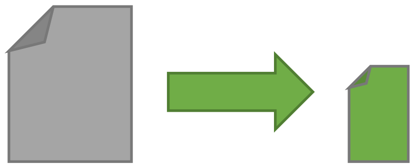
- ***User-provided images are read-only for OS***
  - *read-only layer is stored in the host and shared*
  - *Can be prepared using overlay snapshotter*
  - *Read-only part is Implemented by virtio-fs*
- ***User-generated data does not need to be persisted***
  - *Leveraging reflink copy to build CoW storage.*
  - *Do not persist temporary data to disk.*
  - *Volatile writable layer is implemented by virtio-blk.*



Methodology & Design

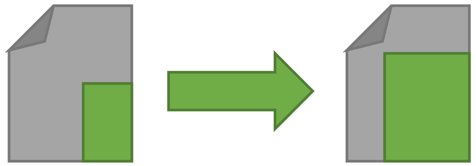
Condensed guest kernel and pre-patched image

- ***Condense the guest kernel to build serverless-customized kernel***
  - *Only retain features required in serverless context*
  - *Without runtime performance degradation*



Reduce kernel size

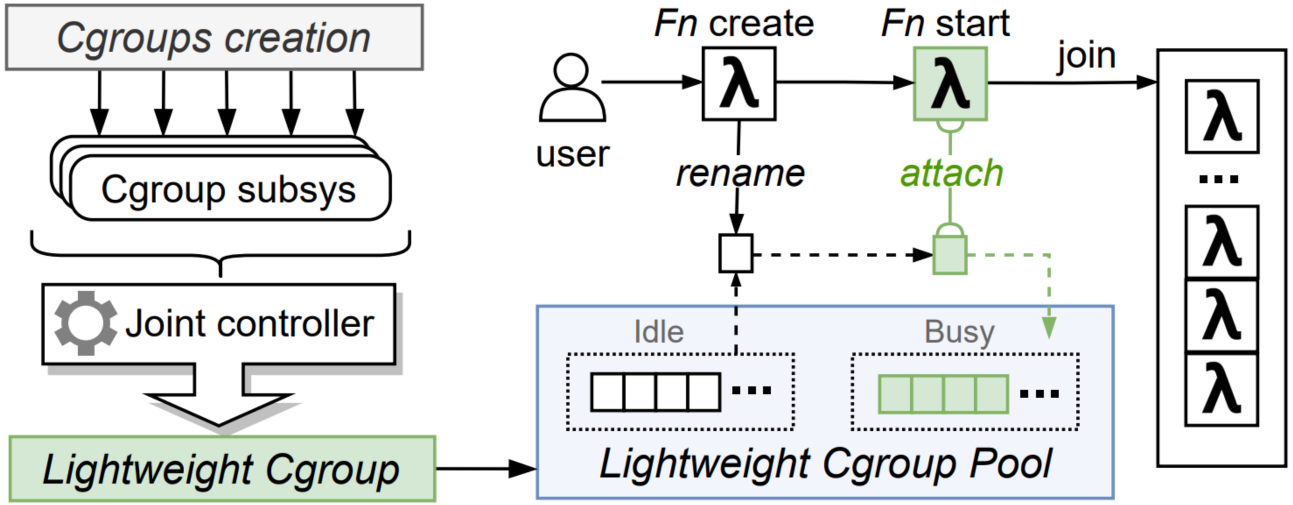
- ***Generate a pre-patched kernel image for template startup***
  - *Re-organizing text/data segments.*
  - *Avoid self-modifying code.*



improve sharable part

Methodology & Design

Lightweight cgroup and cgroup pool



- *The lightweight cgroup aggregates all subsys into one single dedicated one.*
- *"cgroup rename", as a special case, does not need any global lock.*
- *Pre-create and maintain lightweight cgroups in a pool.*

Introduction & Background

Motivation

Methodology & Design

Evaluation

RunD

Evaluation

Evaluation setups:

- Baselines:

  Kata-qemu, Kata-FireCracker, and Kata-template.

- Software and hardware setup:

Table 1: Experiment setup in our evaluation.

		Configuration
Hardware		CPU: 104 vCPUs (Intel Xeon Platinum 8269CY) Memory: 384GB, two SSD drives: 100GB, 500GB
Software		OS: CentOS7, kernel: Linux kernel 4.19.91
Container	kata-qemu	containerd 1.3.10, kata 1.12.1
	kata-FC	containerd 1.5.8, kata 2.2.3
	kata-template	containerd 1.3.10, kata 1.12.1
	RunD	containerd 1.3.10

- Measurement:

  create pod sandboxs without containers inside, through crictl

  smem to collect memory usage

Evaluation

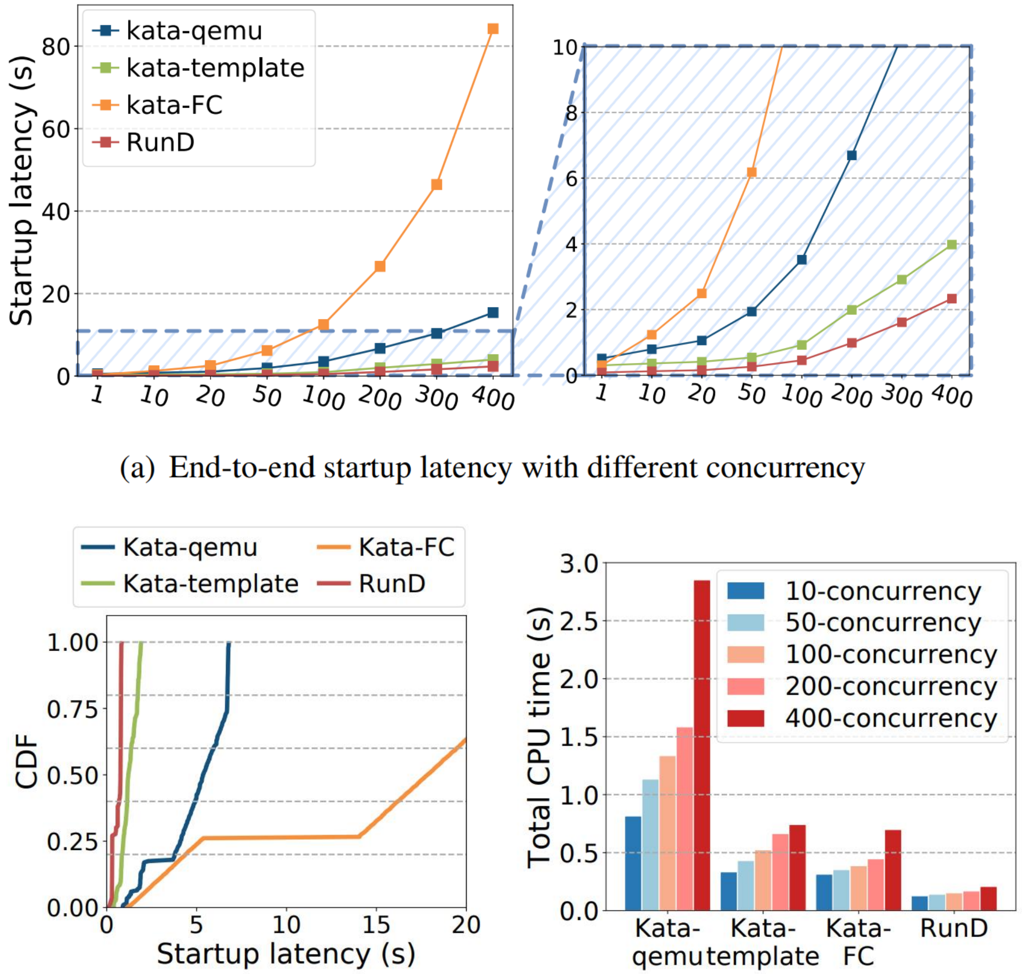
Key improvements:

Avg 88ms

Reduced cold startup latency for a single sandbox

Max 200/s

launch 200 sandboxes simultaneously within 1s, with minor fluctuation and CPU overhead.



(a) End-to-end startup latency with different concurrency

(b) Latency distribution

(c) CPU time

Evaluation

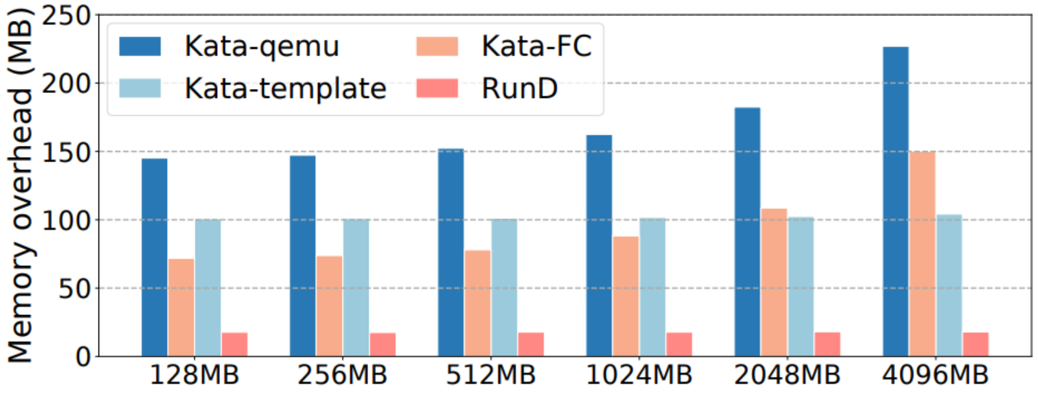
Key improvements:

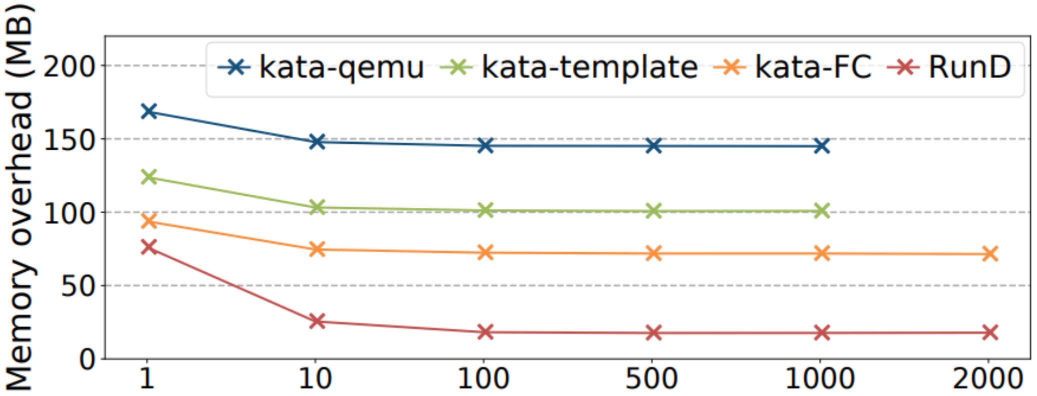
20MB-

The memory overhead is less than 20MB per sandbox with RunD.

2500density

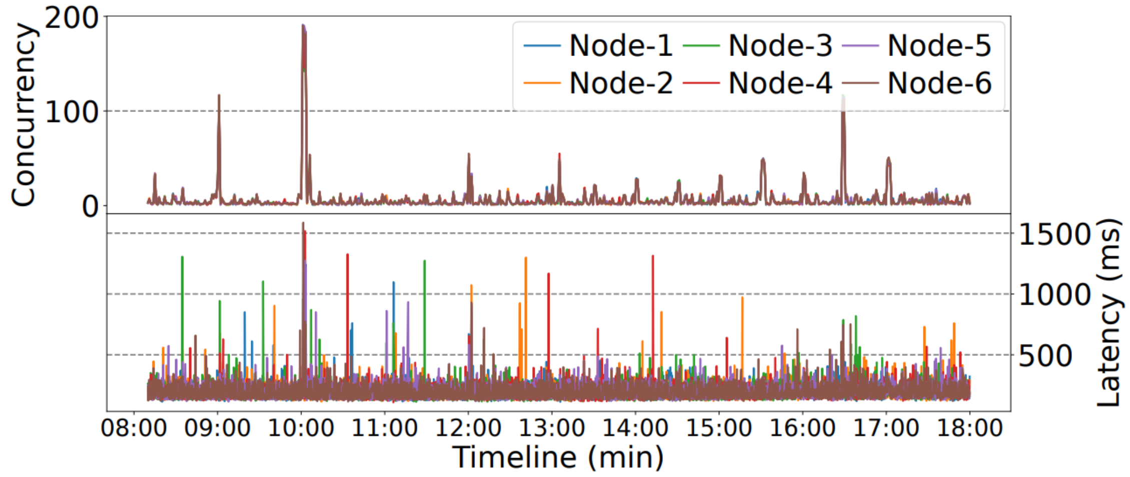
deploy over 2,500 sandboxes of 128MB memory specification on the node with 384GB memory

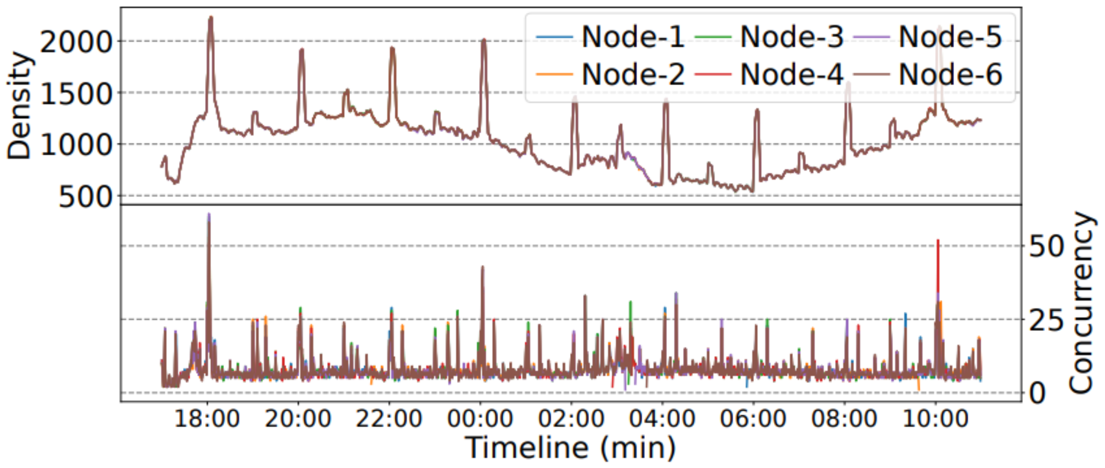




Evaluation

In-production usage for serverless:





Basic guarantee

Low response latency

Two requirements

High-concurrency startup

High-density deployment

Introduction & Background

Motivation

Rationale & Design

Evaluation

Open-Source

RunD Open-source

RunD X OpenAnolis → kata containers

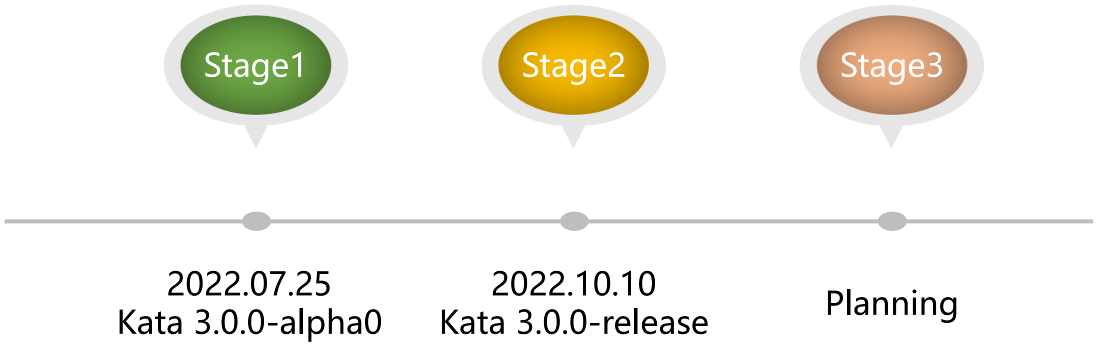
RunD, developed by OpenAnolis Community,
will be open-sourced in the Kata Container Community in October.

RunD guest-to-host solution will drive Kata Container to upgrade from previous version 2.x to version 3.0.

[1] OpenAnolis Community: https://openanolis.cn/?lang=en

[2] Kata Container: https://github.com/kata-containers/kata-containers

RunD (Kata 3.0) Release Plan



Kata version number	Expected release date
3.0.0-alpha0	2022-07-25
3.0.0-alpha1	2022-08-15
3.0.0-alpha2	2022-08-29
3.0.0-rc0	2022-09-12
3.0.0-rc1	2022-09-26
3.0.0-release	2022-10-10

Class	Sub-Class	Development Stage
service	task service	Stage 1
	extend service	Stage 3
	image service	Stage 3
runtime handler	Virt-Container	Stage 1
	Wasm-Container	Stage 3
	Linux-Container	Stage 3
Endpoint	Veth Endpoint	Stage 1
	Physical Endpoint	Stage 2
	Tap Endpoint	Stage 2
	Tuntap Endpoint	Stage 2
	IPVlan Endpoint	Stage 3
	MacVlan Endpoint	Stage 3
	MacVtap Endpoint	Stage 3
	VhostUserEndpoint	Stage 3
Network Interworking Model	Tc filter	Stage 1
	Route	Stage 1
	MacVtap	Stage 3
Storage	virtiofs	Stage 1
	nydus	Stage 2
hypervisor	Dragonball	Stage 1
	QEMU	Stage 2
	Acrn	Stage 3
	CloudHypervisor	Stage 3
	Firecracker	Stage 3

Introduction & Background

Motivation

Rationale & Design

Evaluation

Open-Source

Conclusion

Conclusion

Summary:

- *Read/Write splitting based rootfs mounting.*
  - *Leveraging the read-only and non-persistence features.*
- *Condensed kernel and Pre-patched image with template.*
  - *Reduce the kernel size and improve the sharable part.*
- *Lightweight cgroup and cgroup pool.*
  - *aggregates all subsys into one single dedicated lightweight one, and use “cgroup rename” to avoid serial operations.*

Our next track presentation:

Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing

Proposes to accelerate time-consuming container specialization if it needs cold startup

Thanks!

Q&A

Zijun Li, lzjzx1122@sjtu.edu.cn;

Jiagan Cheng, chengjiagan@sjtu.edu.cn;

Quan Chen, chen-quan@cs.sjtu.edu.cn;

Eryu Guan, eguan@linux.alibaba.com;

Zizheng Bian, zizheng.bian@linux.alibaba.com;

Yi Tao, escape@linux.alibaba.com;

Bin Zha, zhabin.zb@alibaba-inc.com;

Qiang Wang, qw.hust@gmail.com;

Weidong Han, shaokang.hwd@alibaba-inc.com;

Minyi Guo, guo-my@cs.sjtu.edu.cn;