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

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

https://www.usenix.org/conference/atc22/presentation/li-zijun-rund

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

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

[1,2]Zijun Li, [1]Jiagan Cheng, [1]Quan Chen, [2]Eryu Guan, [2]Zizheng Bian, [2]Yi Tao, [2]Bin Zha,
[2]Qiang Wang, [2]Weidong Han, [1]Minyi Guo
[1]*Department of Computer Science and Engineering, Shanghai Jiao Tong University*
[2]*Alibaba Group*

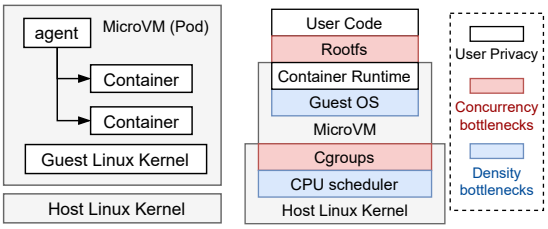
Abstract

The secure container that hosts a single container in a micro virtual machine (VM) is now used in serverless computing, as the containers are isolated through the microVMs. There are high demands on the high-density container deployment and high-concurrency container startup to improve both the resource utilization and user experience, as user functions are fine-grained in serverless platforms. Our investigation shows that the entire software stacks, containing the cgroups in the host operating system, the guest operating system, and the container *rootfs* for the function workload, together result in low deployment density and slow startup performance at high-concurrency. We propose and implement a lightweight secure container runtime, named **RunD**, to resolve the above problems through a holistic guest-to-host solution. With RunD, over 200 secure containers can be started in a second, and over 2,500 secure containers can be deployed on a node with 384GB of memory. RunD is adopted as Alibaba serverless container runtime to support high-density deployment and high-concurrency startup.

1 Introduction

With serverless computing (Function-as-a-Service), tenants submit functions directly to the Cloud without renting virtual machines, and the cloud provider uses containers to host invocations on-demand [26, 31, 35, 48, 48, 56]. Most cloud providers publish the serverless computing services with the pay-for-use pricing model, such as Amazon Lambda [4], Google Cloud Function [11], Microsoft Azure Functions [13], and Alibaba Function Compute [2].

When hosting function invocations, traditional containers (e.g., Docker, LXC) only provide process level isolation [22, 38], as they are implemented based on *Namespace* and *Cgroup*. They cannot prevent privilege escalation, information disclosure side channels, and covert channel communication [20]. To this end, secure containers that achieve the same isolation with the traditional virtual machines are often preferred. MicroVM is for isolation, and the container is for abstraction [30]. Secure container often creates a normal container within the lightweight microVM as shown in Figure 1(a). In such way users can build serverless services based on exsiting container infrastructure and ecosystem. It ensures compatibility with the container runtime in the MicroVM. Kata Containers [19] and FireCracker [20] provide practical experience in implementing such secure containers.



(a) Secure Container model (b) Secure Container stacks

Figure 1: The state-of-the-art secure container model, and several bottlenecks in the architecture stacks.

Figure 1(b) shows the architecture hierarchy of a secure container. In general, the guest operating system (GuestOS) in the microVM and resource scheduling on the host are offloaded to the cloud provider. The *rootfs* is a filesystem and acts as the execution environment of user code. It is created by the host and passed to the container runtime in the microVM. On the host side, cgroups are used to allocate resources to secure containers, and the CPU scheduler manages the resource allocation. The complex hierarchy of secure containers brings extra overhead.

The lightweight and short-term features of functions make high-density container deployment and high-concurrency container startup essential for serverless computing. For instance, 47% of Lambdas run with the minimum memory specification of 128MB [5] in AWS, about 90% of the applications never consume more than 400MB in Microsoft Azure [49]. Since a physical node often has large memory space (e.g., 384GB), it should be able to host many func-

tions. Meanwhile, a large number of function invocations may arrive in a short time. However, the overhead of secure containers significantly reduces the deployment density of functions, and the concurrency of starting containers.

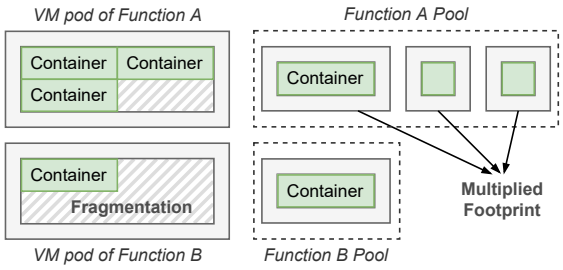
Our investigation identifies two key factors in secure containers that result in low concurrent startup. First of all, *rootfs* either results in unacceptable long latency for writable device provisioning or high CPU overhead under considerable I/O stress, when many containers are started concurrently. Secondly, concurrently starting multiple containers brings a large number of cgroups operations on the host side. However, the cgroup-related operations are serialized in the operating systems. The serialization is due to several mutex locks introduced in the kernel to handle a complex hierarchy of cgroup subsystems. The serial operations slow down cgroups creation for microVMs.

Meanwhile, secure containers amplify the resource overhead of each function, multiply host-side resource consumption with more microVMs, and lower the deployment density. Firstly, for microVMs, the standard Linux kernel is heavyweight for a small-sized memory specification. Secondly, the mainstream block-based solution for container *rootfs* in microVM generates the same page cache in both host and guest, resulting in a duplicated memory overhead. Lastly, CFS (Completely Fair Scheduler) in the host operating system traverses all the cgroups (containers) for balancing the processes, resulting in a significant scheduling overhead at high-density deployment.

We propose and implement a lightweight secure container runtime, named **RunD**, to resolve the above problems through a holistic guest-to-host solution. According to our evaluation, RunD boots to application code in 88*ms*, and can launch 200 secure containers per second on a node. On a node with 384GB memory, over 2,500 secure containers can be deployed with RunD.

The main contributions of this paper are as follows.

1. **Bottlenecks identification in high-density deployment and high-concurrency startup of secure containers in serverless.** We analyze the shortcomings and bottlenecks through a holistic guest-to-host solution, in terms of container *rootfs* storage, the microVM memory footprint, and the overhead of cgroups.
2. **A guest-to-host solution to secure containers for high-density and high-concurrency targets in serverless.** The practice including: 1) a better container *rootfs* implementation based on read/write splitting for serverless; 2) the method to condense the guest kernel and improve kernel sharing by a pre-patched kernel image; 3) the host-side lightweight cgroup design and the rename-based cgroup pool management.
3. **A lightweight serverless runtime RunD for serverless architecture.** We design and open-source RunD based on Kata-runtime, and it shows much higher deployment density and startup concurrency compared with the state-of-the-arts.



(a) Multi-container-per-VM (b) Single-container-per-VM

Figure 2: Two practices of the secure container model.

RunD is adopted as Alibaba serverless container runtime serving more than 1 million functions and almost 4 billion invocations daily. The online statistics demonstrate that RunD enables the maximum deployment density of over 2,000 containers per node and supports booting at most 200 containers concurrently with a quick end-to-end response.

2 Background

In this section, we will discuss the current secure container design, and concerning problems motivating this work.

2.1 Secure Container Models

Based on different levels of security/isolation requirements, there are generally two categories of secure containers in the production environments.

Figure 2(a) shows the *multi-container-per-VM* secure container model that only isolates functions. In the model, a virtual machine (VM) hosts the containers for the invocations of the same function. The containers in the same VM share the guest operating system of the VM. In this case, the invocations to different functions are isolated, but the invocations to the same function are not isolated. Since the number of required containers for each function varies, this model results in memory fragmentations [34]. Though the memory fragmentations can be reclaimed at runtime, it may significantly affect the function performance, and even crash the VM when the memory hot-unplug fails.

Figure 2(b) shows the *single-container-per-VM* secure container model that isolates each function invocation. Current serverless computing providers [1, 20] mainly use this secure container model. In this model, each invocation is served with a container in a microVM. This model does not introduce memory fragmentations, but the microVMs themselves show heavy memory overhead. It is obvious that each microVM needs to run its exclusive guest operating system, multiplying the memory footprints.

The secure container depends on the security model of hardware virtualization and VMM, explicitly treating the guest kernel as untrusted through syscall inspections. With the prerequisite of isolation and security, this work targets the *single-container-per-VM* secure container model.

2.2 Problems with Secure Containers

In production serverless platforms, achieving high container startup concurrency, and high container deployment density are the two key requirements [20]. With the *single-container-per-VM* secure container model, there are problems in achieving the two purposes.

Requirement on high-concurrency container startup. In serverless platforms, each function invocation is short, and a large number of function invocations may arrive in a short time. For example, in Alibaba serverless platform, more than 200 container-launch requests arrive nearly simultaneously on a node. The latency until all containers have entered *main*() can swell super-proportionally due to resource contention among the simultaneously launching VMs. Meanwhile, emerging internet services often show a diurnal load pattern and have bursty loads [18]. A large number of containers are required to be created when the load bursts. Some techniques, such as prewarming containers [31, 42, 49], are able to alleviate container cold startups.

However, bursty loads are inevitable can easily exhaust the limited prewarmed containers. The ability to startup containers at high-concurrency is crucial for serverless platforms.

Requirement on high-density container deployment. The small container specification in a serverless computing platform brings the requirement to deploy containers densely on a node. For instance, 47% of lambda functions run with the minimum memory specification of 128MB in AWS [5]. The actual memory usage of a container may also be smaller than its specifications. As Azure reports [49], about 90% of the applications never consume more than 400MB of memory. A node with 256GB of memory can host $8 \times 256 = 2048$ containers if there is no other overhead. In Alibaba serverless platform, over 2,500 secure containers that 128MB-sized can be deployed on a node with 384GB memory.

Without proactive customizations, secure containers incur extra memory overhead, reducing deployment density in serverless computing. Increasing deployment density greatly improves resource utilization and multi-tenant serving efficiency with the same infrastructure.

3 Problem Analysis and Insights

In this section, we analyze the problems of achieving high-concurrency startup and high-density deployment with secure containers. We use Kata container [19] as the representative secure container to perform the following studies.

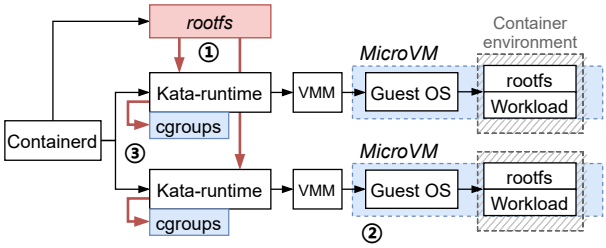


Figure 3: The steps of starting up multiple Kata containers concurrently. The concurrency bottleneck results from creating *rootfs* (step ① in red block) and creating cgroups (step ③ in red flowline). The density bottlenecks result from the memory footprint of the microVM (step ② in blue block) and the scheduling of massive cgroups (step ③ in blue block).

Figure 3 shows the steps of starting Kata containers. First, *containerd* concurrently creates the container runtime *Kata-runtime* and prepares runc-container *rootfs*. Second, the hypervisor loads the GuestOS and the prepared *rootfs* to launch a runc-container in the microVM. Third, the function workload is downloaded into the container and may start to run.

Comparing with starting traditional containers [53], we have two observations when starting up secure containers.

- When starting 100 or more Kata containers concurrently, there is a distinct performance degradation of creating *rootfs* and cgroups, during the Kata-runtime preparation. The degradation results in the low concurrency of starting containers.
- When deploying more than 1,000 Kata containers with 128MB memory specification on a single node with 384GB memory and 104 cores, the microVMs' memory footprint (due to the guest kernel and *rootfs*) already occupies most of the memory space. Meanwhile, the containers' I/O performance is also seriously degraded.

Figure 3 also shows three bottlenecks we found that result in the above two observations. In general, the inefficiency of creating *rootfs* and cgroups results in low container startup concurrency. The high memory footprint and scheduling overhead result in low container deployment density. We analyze the bottlenecks in the following subsections.

3.1 Bottleneck of Container Rootfs Storage

In general, *rootfs* can be exposed to the container runtimes in the microVMs through two interfaces to construct the image layers: filesystem sharing (e.g. *9pfs* [45], *virtio-fs* [16]) and block device (e.g. *virtio-blk* [46]). Filesystem sharing enables microVMs to access a directory tree on the host directly. When the block device is used, the host creates block devices through the device-mapper [8] and passes them to the microVMs, so that containers can access data at the block level, rather than the file level.

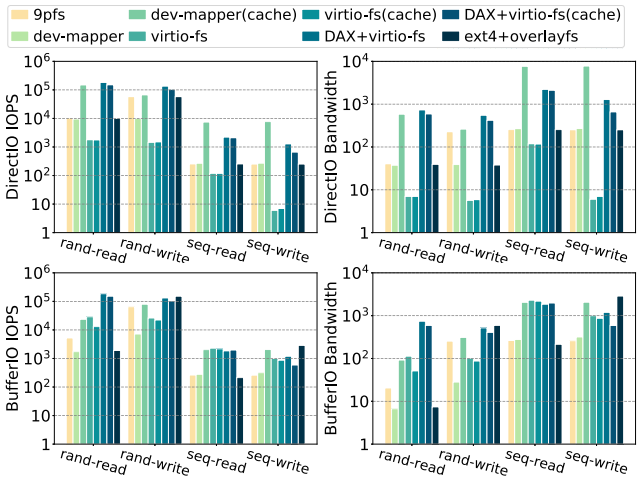
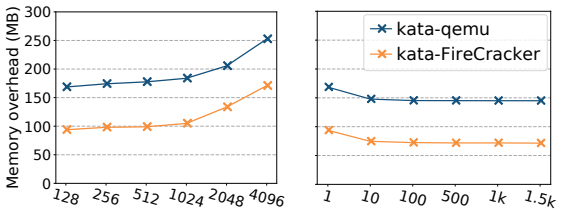


Figure 4: The IOPS/bandwidth performance of rand/seq directIO/BufferIO read/write when using different *rootfs* mapping in Kata-runtime (dev-mapper represents that *virtio-blk* is used, *ext4+overlayfs* represent the baseline of default runc-container *rootfs* implementation).

Figure 4 shows the IOPS (IO-Per-Second) and IO bandwidth of the random/sequential read and random/sequential write, when Kata uses *9pfs*, *virtio-fs*, and *virtio-blk*, respectively. We also measure the metrics of using *ext4* file system and *overlayfs* file access interface on the host node, to denote the case of the traditional containers [50, 51]. As observed, microVMs should not use *9pfs* as *rootfs* storage interface due to the poor performance.

With the default configuration (cache enabled), *virtio-blk* performs best at random/sequential writing. However, the device-mapper who prepares the block device in the host cannot meet the high-concurrency requirement [59]. According to our measurement, it takes as high as 10 seconds to prepare a *rootfs* when 200 containers are started concurrently, while it only takes about 30 milliseconds for a single container startup. In this case, the operation of preparing *rootfs* timeouts, resulting in the container breakdown. Moreover, *virtio-blk* inherently does not support the page cache sharing between host and guest operating systems. When *virtio-blk* backend reads *rootfs* files into the host page cache, the mapped content reproduces the same page cache in the guest. The issue of duplicated page cache brings a high memory footprint overhead.

Virtio-fs resolves the problem of duplicating page cache. When *DAX* is enabled in *virtio-fs*, it allows bypassing guest page cache and mapping host page cache directly in guest address space [16]. However, *virtio-fs* results in poor random/sequential write performance (Figure 4). In addition, each container has to employ a client daemon to support *virtio-fs* I/O operations, leading to excessive CPU usage when enormous containers colocate. Things get worse for



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

Figure 5: The memory overhead of a secure container.

either large I/O stress under high-concurrency or massive operations of metadata processing.

The above investigation shows that either virtio-fs or virtio-blk can compromise either deployment density or startup concurrency of secure containers. An exploratory alternative would be: using virtio-fs to support the read-only part of rootfs for sharing page cache between host and guests, and using virtio-blk to support the writeable part of rootfs for high I/O performance. A solution is also required to further reduce the duplicated writable part for rootfs.

3.2 High Memory Overhead Per Container

Except for the memory used by the user function, the memory footprint of other components in the secure container is the memory overhead. The 5MB memory overhead reported in FireCracker [52] is the overhead of the FireCracker VMM itself. In the microVM of a secure container, the guest operating system, the struct page for memory management, and other components (e.g., baseOS, shimv2, agent) also consume additional memory space [52].

Figure 5 shows the per-container memory overhead of secure containers with different memory specifications and at different deployment densities. In the figure, Kata-qemu is the secure container that uses *qemu* as the hypervisor, and Kata-FireCracker uses *FireCracker* as the hypervisor. As observed in Figure 5(a), the memory overheads of a 128MB container are 94MB and 168MB with Kata-FireCracker and Kata-qemu, respectively. The overhead increases with the memory specification of the container.

The average memory footprint of a single microVM can be reduced by sharing the text/rodata segment among multiple microVMs. Mainstream MicroVMs achieve it by mapping the kernel file to the guest memory directly using *mmap*. As shown in Figure 5(b), the per-microVM memory overhead of *kata-qemu* and *kata-FireCracker* reduce to 145MB and 71MB when 1,000 VMs are deployed on a node. However, the overhead is still too large for a serverless container with only 128MB memory specification.

MicroVM template (e.g., Kata template) [17, 29, 52] is a popular method to further reduce the per-microVM memory overhead, while preserving microVM consistency. The

template serves as a primary image for a microVM copy that includes disks, devices, and settings. New microVMs are created by on-demand forking from a pre-created template microVM, and text/rodata segments are also shared among multiple microVMs [54] in read-only mode. The unaccessed kernel files of the template will not consume the physical memory, reducing the memory overhead.

However, the template technique is not as efficient as we thought, due to the self-modifying codes in the operating system kernel [24, 25]. The self-modifying code technique alters the instructions on-demand as it runs, and the Linux kernel relies heavily on self-modification code to improve performance on boot and during runtime. We start a clean microVM with CentOS 4.19 guest kernel from a template to investigate the impact of self-modifying codes. The investigation shows that 10,012KB of the code and the read-only data is accessed in the memory, but 7,928KB of them were modified during boot. This case in point reveals that the self-modifying codes degrade the efficiency when using *mmap* for less memory consumption of kernel image files.

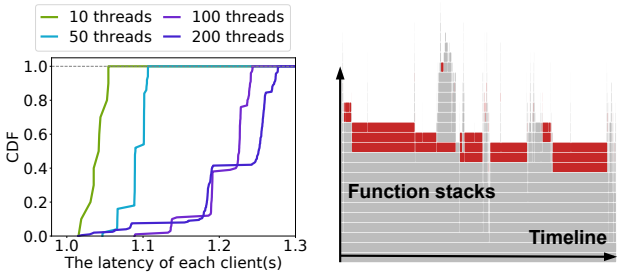
The code self-modifying reduces the shareable memory when using microVM template. Reducing the self-modifying codes in the guest kernel is worth investigating if they are not necessary for the serverless computing scenario.

3.3 High Host-side Overhead of Cgroups

Cgroup is designed for resource control and abstraction of processes. In serverless computing, the frequency of function invocations shows high variation. In this case, the corresponding secure containers are frequently created and recycled. For instance, in our serverless platform, at most 200 containers would be created and recycled on a physical node concurrently in a second. The frequent creating and recycling challenge the cgroup mechanism on the host.

We measure the performance of cgroup operations when creating 2,000 containers concurrently. In the experiment, we use different numbers of threads to perform cgroup operations. Figure 6(a) shows the cumulative distribution of container creating latencies. Counter-intuitively, the latency increases when more threads are used, even if each thread needs to create fewer containers.

The reason behind the above fact is that the Linux kernel introduces several global locks (e.g., *cgroup_mutex*, *css_set_lock*, *freezer_mutex*) to serialize cgroup operations. The global locks are used to coordinate more than 10 resource subsystems (aka. the cgroup subsys) involved in cgroup. Figure 6(b) shows the flame graph of creating 2,000 cgroups using 10 threads concurrently. In the figure, the red parts show the case that "mutex locks" are active. When the cgroup mutex uses the optimistic spinning by default, the spinner cgroups experience the optimistic spinning if they fail to acquire the lock. It will lead to heavy CPU consumption and belated exiting of the critical section in the multi-



(a) Latency distribution (b) The flame graph of Perf

Figure 6: The performance of cgroup operations when creating 2,000 containers concurrently. Due to the mutex lock, the cgroup operations have higher latencies as the concurrency increases.

threaded scenarios. *Therefore, the locks serialize the operations of cgroups and drag down the latencies.*

Besides, a common observation is that there are often more than 10,000 cgroups with thousands of containers on a compute node. The PELT (Per-Entity Load Tracking) for load balancing in CFS will iterate over all cgroups and processes when scheduling these containers. In this scenario, the frequent context switching and hotspot functions that involve high-precision calculation in the scheduler become a bottleneck, accounting for 7.6% of the CPU cycles of the physical node, according to our measurement.

The host-side overhead of cgroups prohibits the high-density deployment and high-concurrency startup in serverless computing. Simplifying the cgroup design, and reducing the critical section introduced by the mutex locks, are fundamental solutions to eliminate the high host-side overhead.

4 Methodology of RunD

The above analysis reveals the bottlenecks in the host, the microVM, and the guest in achieving the high-concurrency startup and high-density deployment. We propose **RunD**, a holistic secure container solution that resolves the problem of duplicated data across containers, high memory footprint per VM, and high host-side cgroup overhead.

In this section, we first show a general design overview of RunD, and then present the design of each component to resolve the corresponding problem.

4.1 Design Overview

When designing RunD, we have a key implication for serverless runtime. The negligible host-side overhead in a traditional VM can cause amplification effects in the FaaS scenario with high-density and high-concurrency, and any trivial optimization can bring significant benefits. Utilizing the

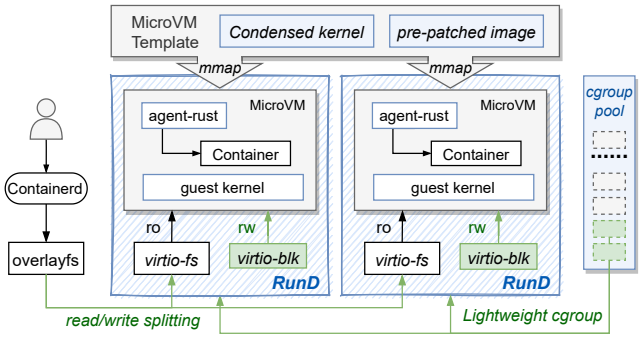


Figure 7: The lightweight serverless runtime of RunD. The condensed kernel and pre-patched image land in the guest domain, while read/write splitting-based *rootfs* and a lightweight cgroup pool land in the host domain.

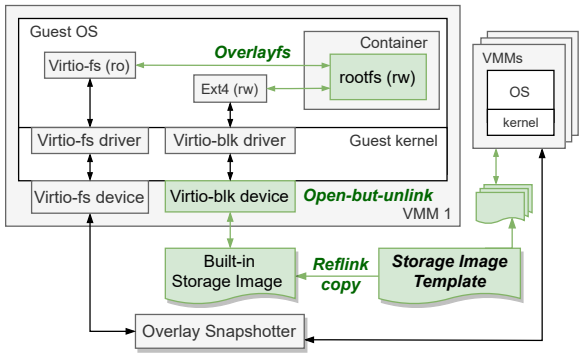


Figure 8: The read/write splitting of container *rootfs*. *Virtio-fs* is used to handle the read-only layer and *virtio-blk* is used to handle the volatile writable layer.

features of read-only data/runtime and non-persistent storage in serverless, RunD proposes guest-to-host solutions.

Figure 7 shows the RunD design and summarizes our methodologies. RunD runtime makes a read/write splitting by providing the read-only layer to *virtio-fs*, using the built-in storage file to create a volatile writeable layer to *virtio-blk*, and mounting the former and latter as the final container *rootfs* using *overlayfs*. RunD leverages the microVM template that integrates the condensed kernel and adopts the pre-patched image to create a new microVM, further amortizing the overhead across different microVMs. RunD renames and attaches a lightweight cgroup from the cgroup pool for management when a secure container is created.

Based on the above optimizations, a secure container (referred to as a "sandbox") is started in the following steps, when RunD is used as the secure container runtime.

- In the first step, once *containerd* receives a user invocation, it forwards the request to RunD runtime.
- Second, RunD prepares the runc-container *rootfs* for the virtual machine hypervisor. The *rootfs* is separated into read-only layer and writable layer. (Section 4.2).
- Third, the hypervisor uses the microVM template to create the required sandbox (Section 4.3), and mount the runc-container *rootfs* into the sandbox by *overlayfs*.
- Lastly, a lightweight *cgroup* is attached to the sandbox (Section 4.4), to manage the resource allocation for this sandbox in the host.

4.2 Efficient Container Rootfs Mapping

Section 3.1 examines the challenges in the high-density and high-concurrency scenario for container *rootfs*. The current secure container fails to discriminate between serverless platforms and traditional infrastructure-as-a-service environments. The mainstream solutions are designed for persistent data storage, and it is the key point why container *rootfs* storage imposes restrictions on our goals.

We investigate the data in a sandbox in the serverless computing scenario, and find that **user-provided code/data is read-only for the operating system, and the system-provided runtime files are also read-only for user functions.** Meanwhile, the data in the local memory or storage generated in a sandbox will not be used by subsequent function invocations, due to the stateless feature of serverless computing. The temporary and intermediate data generated during the function execution is not required to be persisted.

Based on the above finding, it is possible to split the *rootfs* into a read-only layer and a writable layer, and then handle them in different ways [32]. The sandboxes can share the read-only layer on the same node, and the writable layer has to be prepared separately for each sandbox.

Figure 8 shows the way to split *rootfs* into a read-only layer and a volatile writable layer. According to the investigation in Section 3.1, *virtio-fs* is used to handle the read-only layer, and *virtio-blk* is used to handle the volatile writable layer for better performance. The read-only layer is stored in the host and can be prepared in negligible time when using the overlay snapshotter provided by the container runtime. However, it is challenging to handle the volatile writable layer efficiently. By default, the host operating system needs to prepare a logic storage volume for the sandbox. This operation is time-consuming and is one of the most important reasons that result in the long latency of preparing *rootfs*.

We propose the volatile block device as the volatile writable layer, considering the volatile feature of the writable layer in serverless platforms. The volatile block device will not persist temporary data from user functions to the disk, unlike the logic storage volume. A *storage image template* is pre-created in the host as the base file. When creating a volatile block device for a new sandbox, a *build-in storage image* is created and linked to the *storage image template*, using *reflink* [60]. *reflink* enables *storage image template* to

share data with *build-in storage images* in a CoW (Copy-on-Write) fashion. Then a volatile block device is created associated with the *build-in storage image*. Once the hypervisor opens the device, the *build-in storage image* will be deleted. The volatile block device ensures that user functions can perform writing as usual without persisting data on the local disk.

We compare our solution with the traditional ones in which the entire *rootfs* is created by the device-mapper as a block device. When 200 sandboxes are started concurrently, traditional solutions incur 4,500 IOPS and use 100MB/s IO bandwidth. On the contrary, our solution incurs only 1,500 IOPS and uses 8MB/s IO bandwidth. Better, the time needed to prepare the *rootfs* decreases from 207ms to only a negligible 0.2ms, and the writing performance of our solution is the same as that of the mainstream transmission.

4.3 Condensed and Pre-patched Guest Kernel

In this subsection, we present two techniques used to reduce the memory used by each sandbox, so that the deployment density can be significantly increased.

4.3.1 Reducing the guest kernel size

Following the abstraction premise in current serverless platforms, the guest environment management for serverless containers is offloaded to the cloud provider. Meanwhile, RunD depends on the security model of hardware virtualization and VMM, explicitly treating the guest kernel as untrusted through syscall inspections. Based on this fact, there is an opportunity to condense the guest kernel for the lightweight characteristic of serverless functions. Considering that several features in the guest kernel are redundant and memory intensive in the serverless context, RunD condenses these features at compile-time. When customizing the condensed guest kernel, the principles behind it are as follows:

- Minimize kernel memory footprint and image size.
- Retain features required in the serverless context.
- Without runtime performance degradation.

Following the above principles, we build the condensed kernel for the guest operating system based on Linux kernel, by disabling features:

- Do not pre-create loop device (2.2MB Mem reduced).
- Disable *acpi* and *ftrace* (2MB and 6MB Mem reduced).
- Disable *graphics*-related items (2MB Mem reduced).
- Disable *i2c* and *ceph* (3MB Mem reduced, and 4MB reduced of kernel image size).
- Kernel files (560K Mem and 571K image size reduced).

Validating all features at compile-time case by case, RunD effectively reduces the memory footprint of a CentOS 4.19 Linux kernel by about 16MB and condenses the kernel image by about 4MB. Based on this condensed guest kernel, we review several investigations of the self-modifying code and propose our solution to reduce the memory overhead further.

4.3.2 Alleviating code self-modification

As mentioned before, cloud providers manage and maintain the underlying hardware and execution runtimes in serverless context, standing for that all microVMs on the same node generally use the same guest kernel. In this scenario, the sandboxes on the same node generate the same patched kernel code, even if they execute the self-modification patch logic. This is because **the self-modifying code of kernel text segments only occurs at the startup phase, after which the kernel code area becomes "read-only after initialization".** In this case, sandboxes experience the same initialization phase and generate predicable self-modifying code segments.

Based on the above observation, there is an opportunity to generate a pre-patch guest kernel image file already patched with self-modified code segments. The MicroVM template technique discussed in Section 3.2 may work efficiently without self-modifying code.

Adapting to this optimization, we also resolve the potential kernel panic issues when loading the pre-patched kernel image for higher stability. RunD tries to share as many kernel files as possible across different secure containers. With a pre-patched microVM template, RunD not only reduces the memory footprint of a single container for higher-density deployment, but also allows to quickly fork multiple instances [29, 52].

4.4 Lightweight Cgroup and Cgroup Pool

In Section 3.3, we analyze that serialized cgroups operations in the host become one of the bottlenecks of secure containers with high-density deployment and high-concurrency startup. The intuitions are to efficiently handle synchronization access on mutex structures and reduce the number of cgroups with a better design.

Our further investigations reveal the optimization opportunities in two aspects. Firstly, creating containers involves multiple cgroup subsystems (e.g. *cpu*, *cpuacct*, *cpuset*, *memory*, and *blkio*). Because the Linux kernel cannot parallelize these cgroup-related operations, creating these groups for each sandbox is time-consuming. Secondly, **pre-creating and maintaining cgroups in a pool can effectively reduce the creation overhead, since afterward only the `cgroup rename` is used.** The `cgroup rename`, as a special case, is a lightweight operation without acquiring any global lock. Following these two observations, we propose a lightweight cgroup and the cgroup pool, as shown in Figure 9.

The lightweight cgroup decreases the total number of cgroups and system calls. Rather than creating the cgroup for each subsystem, we aggregate necessary cgroup subsystems

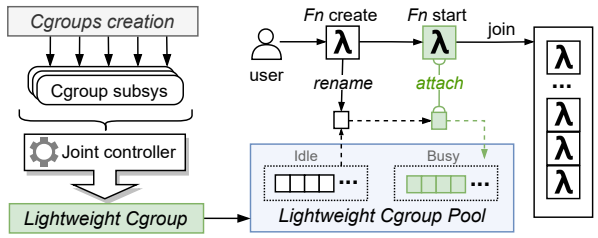


Figure 9: The lightweight Cgroup aggregates all subsystems, eliminating the time-consuming creation by renaming from the Cgroup pool.

(aka the *cpu*, *cpuacct*, *cpuset*, *memory*, and *blkio*) into one single dedicated lightweight cgroup. The implementation of the joint cgroup controller helps RunD reduce the redundant cgroup operations when a container is started, significantly decreasing the total number of cgroups and system calls.

The cgroup pool with renaming mechanism eliminates the time-consuming cgroup creation and initialization. RunD pre-creates corresponding lightweight cgroups and maintains them in a cgroup pool based on the pre-defined node capacity. These cgroups are marked idle when initialized, and are protected in a linked list. For each created container, RunD simply allocates an idle cgroup, updates the state to busy, performs the `cgroup rename` operation, and then attaches the container to this renamed cgroup when a container is started. If a container triggers recycling, RunD will take the cgroup back to the pool, kill the corresponding instance process, and then update the returned cgroup state to idle for subsequent allocating and renaming.

Adopting the above optimizations in kernel mode, we replay the evaluation in Section 3.3. The cgroups creation only consumes 0.09*s* (1 thread), 0.1*s* (50 threads), and 0.14*s* (200 threads), respectively. Compared with the default mechanism, the lightweight cgroup and the rename-based cgroup pool reduce 94% of the cgroups creation time.

5 Evaluation

In this section, we evaluate the performance of RunD in supporting high-concurrency startup and high-density deployment of secure containers, and introduce the performance of RunD in production usage.

5.1 Evaluation Setup

We have implemented and open-sourced RunD with Rust, a more memory-efficient and thread-safe programming language. RunD runtime involves four main modules: Containerd-shim (21k LOC), Device (4.4k LOC), Hypervisor (5.6k LOC), and Lightweight-cgroup (20k LOC).

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

Baselines: we compare RunD with the state-of-the-art secure container, Kata Containers [19]. Specifically, we use three popular configurations of Kata containers: *Kata-qemu*, *Kata-template*, and *Kata-FC*. *Kata-qemu* uses QEMU [15, 23] as the microVM hypervisor, *Kata-template* uses QEMU while integrating container template, *Kata-FC* uses lightweight FireCracker [20] as the microVM hypervisor. Kata-qemu and kata-template use an old version of Kata Containers, as the new version has some bugs that result in poor performance. Table 1 shows the detailed setups.

Testbed: we run the experiments on a node with 104 virtual cores, 384GB memory, and two SSD drives of 100GB and 500GB. Such specification is widely-used in production clouds. The 100GB drive is used as the root filesystem of the host operating system, and the 500GB drive is used by the secure containers. We use Alibaba Cloud Linux 2 for RunD and Alpine Linux [3] for others, as the guest operating systems in the microVM for a low memory footprint.

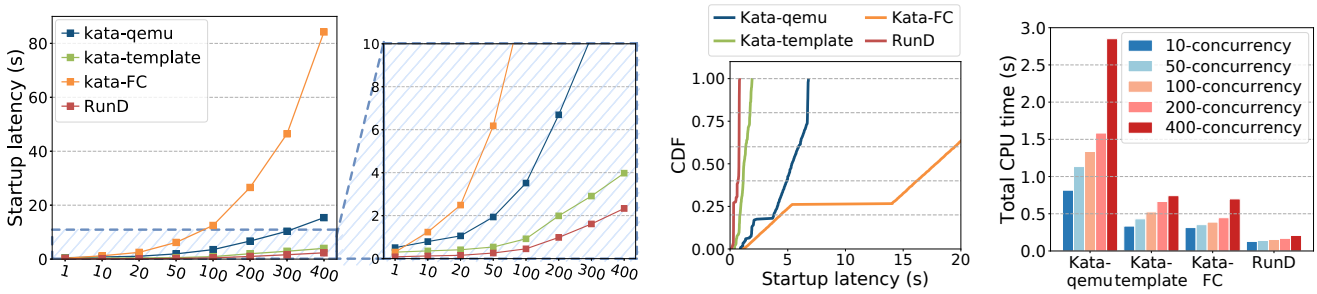
Measurement: in the CRI specification [6], a pod sandbox refers to a microVM with a lightweight pause container [12]. In all the tests, we only create the pod sandboxes without other containers inside, through the `crictl` command. In the following evaluations, the memory specification of a container denotes the size of memory that can be used by itself. The actual memory usage of a container is collected using the `smem` command.

As RunD is proposed to maximize the supported container startup concurrency and deployment density, in the experiment, we start empty secure containers without user codes or data considering that it is a common practice in FaaS to start empty containers concurrently for prewarming. The in-production results show the performance of RunD for actual workloads with all the steps involved.

5.2 Concurrent Startup Measurement

In this experiment, we focus on three critical metrics related to user experience: *(1) the time needed to start a large number of sandboxes concurrently*, *(2) the startup latency distribution of the sandboxes*, and *(3) the CPU overhead on the host*. The first metric reveals the throughput of starting sandboxes, and the second metric reveals the experience of every user.

As for the first metric, Figure 10(a) shows the time needed to start a large number of sandboxes concurrently. In the fig-



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

(b) Latency distribution

(c) CPU time

Figure 10: The startup metrics with different runtime and concurrency: (a) The end-to-end latency of concurrent startups. The right figure is an enlargement of the left one ($y \in [0, 10]$). (b) The CDF of startup latencies from a 200-way concurrent launch. (c) The CPU usage of concurrent startups.

ure, the *x*-axis shows the number of sandboxes to be started concurrently, the *y*-axis shows the overall time needed to startup all the sandboxes.

As shown in the figure, RunD uses the shortest time to start a large number of sandboxes for all concurrency levels. When 200 containers are created concurrently (we already observe such high-concurrency in Alibaba serverless platform), Kata-FC, kata-qemu, kata-template, and RunD needs 47.6*s*, 6.85*s* and 2.98*s* and 1*s* to create them. Kata-FC requires a much longer time to startup the sandboxes when the concurrency is high. This is because Kata-FC uses *virtio-blk* to create *rootfs*, and the performance is poor at high-concurrency, as we measured in Section 3. There is no such bottleneck in Kata-template and Kata-qemu. Kata-template simply uses template to reduce the overhead of guest kernel and *rootfs* loading, but the inefficient *rootfs* mapping, code self-modification and high host-side overhead of the cgroup operations still exists. As a result, it performs worse than RunD at high startup concurrency. The overall optimizations suggest that RunD provides the performance improvement of about 40% over its nearest baseline, Kata-template, at high-concurrency (e.g., 400-way) startup.

As for the second metric, Figure 10(b) shows the latency distribution of starting each sandbox, when 200 sandboxes are started concurrently. RunD and Kata-template are able to start sandboxes in a stable short time, but the latencies of starting sandboxes with others are out of expected. Users can have identical good experiences with RunD.

As for the CPU overhead, Figure 10(c) shows the CPU time needed on the host to startup sandboxes. When the concurrency is high, RunD greatly reduces the CPU overhead. For instance, when 200 sandboxes are started concurrently, RunD reduces 89.3%, 74.5% and 62.1% CPU overhead compared with Kata-qemu, Kata-template, and Kata-FC, respectively. In addition, the CPU overhead of RunD only increases slightly, when the concurrency increases. This is due to the read/write split policy and the reduction of compute-intensive operations in cgroups. Therefore, RunD is scalable in starting more sandboxes concurrently.

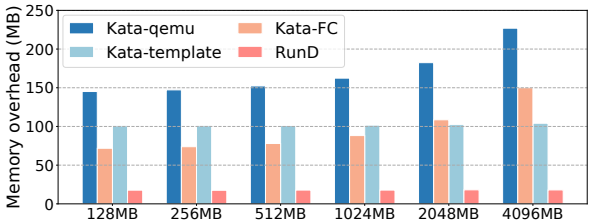


Figure 11: The memory overhead of Kata-qemu, Kata-template, Kata-FC, and RunD (100 sandboxes are deployed).

*In summary, RunD is able to start a single sandbox in **88ms** and launch **200** sandboxes simultaneously within **1s**, with minor latency fluctuation and CPU overhead.*

5.3 Deployment Density

In this experiment, we evaluate the effectiveness of RunD in increasing the sandbox deployment density. In general, the memory used by each container determines the deployment density, while the CPU time needed by each function invocation is minor in the serverless platform. Figure 11 shows the memory overhead when 100 sandboxes are deployed on the experimental node. In the figure, the *x*-axis shows the memory specification of each sandbox.

As observed, RunD has the least memory overhead among four runtimes, and does not increase with the memory specification. The memory overhead is less than 20*MB* per sandbox with RunD. Compared to kata-qemu, kata-template and kata-FC, the overhead of RunD is reduced by 54.9%, 27.2%, and 18.9%, respectively, even when the memory specification is 128MB. The memory overhead does not increase, because the microVM template technique uses the on-demand memory loading for the containers. Therefore, the page table required for memory management is determined by the actually used memory space. On the contrary, the memory overheads introduced by Kata-qemu and Kata-FC increase

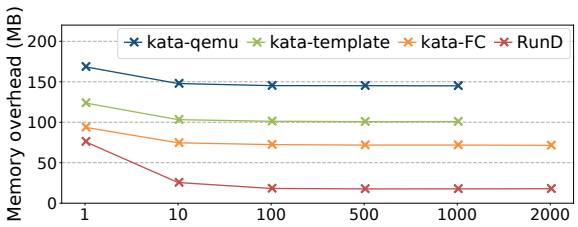


Figure 12: The memory overhead and the amortization by multiple secure containers. The missing point around 2,000 indicates the over-subscription for physical memory space.

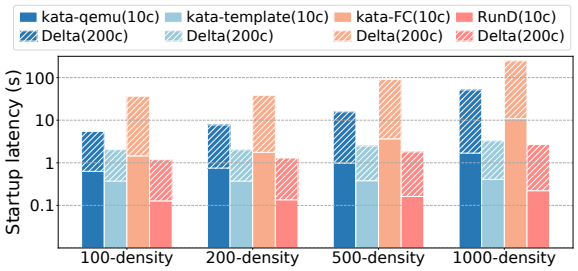


Figure 13: The end-to-end startup latency at different deployment densities. (10c/200c means a 10/200-way concurrent startup, and the Delta means the overhead increment compared with a 10-way concurrent statrup).

with larger memory specifications, as the page table is built for all available memory. In addition, the pre-patched kernel image in RunD further reduces memory overhead.

Figure 12 shows the average memory overhead of the sandboxes when different numbers of sandboxes are deployed on a node. The *x*-axis shows the deployment density. As observed, the average memory overhead reduces with the deployment density, as the sandboxes share the mapped code/data segments. RunD reduces the memory overhead by 87.7%, 82.4%, and 75.1% when 1,000 sandboxes are deployed, respectively, compared with kata-qemu, kata-template, and kata-FC.

*RunD supports to deploy over **2,500** sandboxes of 128MB memory specification on the node with 384GB memory.*

5.4 Impact of Deployment Density on Startup Latency and Concurrency

When some sandboxes are already deployed on a node, the performance of starting sandboxes concurrently is affected. Figure 13 shows the time needed to boot 10 and 200 sandboxes, when some sandboxes are already deployed on the node. The *x*-axis shows the number of already deployed sandboxes. The *y*-axis is in the log10 scale.

When 1,000 sandboxes are already deployed, the time needed to startup 10 containers increases by 1.69*s*, 0.41*s*, 10.8*s*, and 0.22*s* compared with the cases in Figure 10(a) with Kata-qemu, Kata-template, Kata-FC, and RunD. In addition, the time needed already increases with the number of already deployed sandboxes.

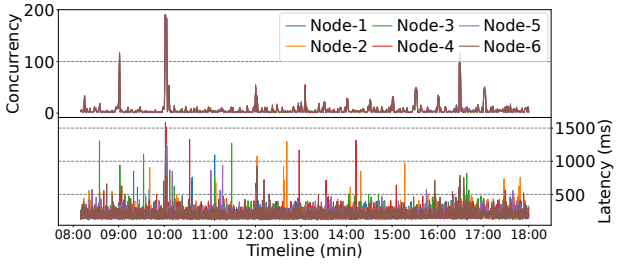


Figure 14: The startup latency and concurrency tracing of RunD in Alibaba serverless platform.

We can also observe that, the time needed to start 200 sandboxes is at least 10 times as much as that needed to start 10 sandboxes at a 1,000-density deployment in all the tests. The significant increase originates from a large number of cgroups in the host operating system. Scheduling and managing containers with these cgroups consume more CPU cycles, thus resulting in CPU bottlenecks appearing earlier than a low-density deployment. The increased time is the smallest with RunD, because it already eliminates many time-consuming cgroup operations.

RunD shows better performance and stability in supporting high-concurrency startups at high-density deployment.

5.5 In-Production Usage for Serverless

Currently, Alibaba serverless computing platform has adopted RunD. The platform serves almost 4 billion invocations from more than 1 million different functions per day.

Figure 14 reports the sandbox startup concurrency and the corresponding startup latency from six nodes. The specification of each node is the same as our experimental setup in Table 1. The data is collected between 08:00 and 18:00 of Jan 10^{th}, 2022. There are about 800 active sandboxes on each node, when the concurrency data is collected. The in-production startup latency of sandboxes at high-concurrency is consistent with that reported in Section 5.4.

As observed from the figure, the startup concurrency bursts at the beginning of each hour. At most 191 sandboxes are started concurrently around 10:00. RunD starts the 191 sandboxes in 1.6 seconds. We look into the function invocation logs, and find that the periodic burst is caused by the an-hour time trigger and cluster-level load balancing. The periodical burst is pervasive, as the Azure serverless platform traces [14] show the same pattern. In the figure, the sandbox startup latency occasionally increases when the concurrency is low. The long time results from the operation in loading large-scale workloads from the tenants. Although the startup

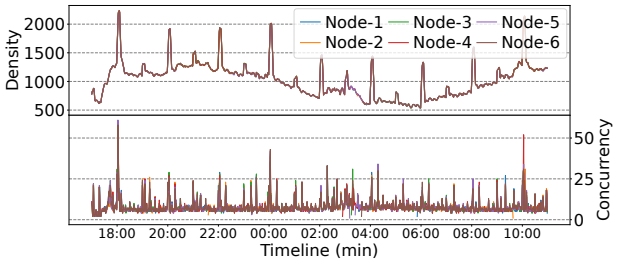


Figure 15: The deployment density and concurrency of RunD in 1 minute intervals, from Alibaba production traces.

concurrency is not always high, it is crucial to ensure a quick startup for a good user experience.

Figure 15 shows the deployment density of the sandboxes on each node. We collect the density statistics of the six nodes between 18:00 of Jan 10^{th} to 10:00 of Jan 11^{th}, 2022. As observed, more than 2,000 sandboxes are deployed on a node at most. We can also find that the high-density deployment happens at the same time as high concurrent startups. This is because many tasks are triggered at the beginning of each hour. The deployment density does not achieve the theoretical upper limit of $384GB/(128{+}20)MB{=}2656$ containers, as some functions use more than 128MB memory, and the workloads are also balanced to other nodes.

RunD is production-verified to meet the high-concurrency startup and high-density deploymant requirements.

5.6 Lessons Learned from Production Usage

Besides the RunD secure container, we have some insights about designing secure containers for serverless systems.

Lesson-1: the CRI specification designed for Kubernetes is not suitable for serverless system. In CRI, multiple related containers can co-locate in the same sandbox, and a lightweight pause container is started first to prepare the cgroups for the remaining containers. This pause container-based solution is negative for serverless computing, as each sandbox only has a single container for security and privacy.

Lesson-2: Functions tend to use the same standard guest environment provided by the serverless platform. In this case, the language environment (e.g., JVM) can also be integrated into the microVM template. However, the language-level template will invalidate the on-demand memory loading in the guest because some language runtimes need to pre-allocate the available memory. There are tradeoffs between higher memory utilization and less startup time, and the decision should be made based on how often the functions share the language environment.

Lesson-3: The memory usage of user functions is the key aspect determining the upper limit of the deployment density. Most functions are lightweight. When 2,000 sandboxes are deployed on a node of our serverless platform, the CPU utilization does not achieve 50%, and there is no complaint on the poor performance from users. One reason for the low CPU utilization is that many sandboxes are actually idle and "kept-alive" after its function invocation is completed in a serverless scenario.

6 Related Work

The most closely related work to RunD is FireCracker [20], which proposes a lightweight VMM for serverless runtime. It provides fast startup within 125ms, allowing 150 VMs to start concurrently per second per node, with less than 5MB footprint per VM. However, FireCracker only serves as the hypervisor stack in the Security Container model, without other complex related processes, e.g., *rootfs* [52]. By contrast, RunD investigates the guest-to-host solution through all stacks and provides higher concurrency and density.

Higher-density deployment. Regarding serverless computing, in the space of higher function deployment density of Secure Containers and VMs [57], the key is designing a more lightweight container runtime both in guest and host. Unikernel [36, 37, 43, 47] runs as a built-in GuestOS without necessary add-ons, demonstrating great potential for deploying containers with less overhead. Kuo [33] Explores lightweight guest kernel configurations for use in Unikernel environments, which has similarity to the approach towards reducing guest kernel size. However, Unikernel is hard to be changed once after compilation with the application. Its compile-time invariance results in poor flexibility in practice. SAND [21] adopts the *multi-container-per-VM* model to amortize the memory footprint of sandboxing. However, they do not further investigate the utilization impact of memory fragmentations in a real-system with high-density deployment. Gsight [61] observes that fine-grained function-level profiling can expose more predictability system-level features in the partial interference. With a more accurate interference predicting [27, 44], the function density can get improved with QoS guaranteed.

The above studies make sense in improving the effective density with less interference for serverless. They are orthogonal to our work, because RunD is motivated to improve the maximum deployment density on a signe node.

Higher-concurrency startup. In the space of higher function startup concurrency, recent approaches leverage the container prewarm pool [9, 40, 49, 58]. The state-of-the-art on container prewarming, SOCK [42], uses a benefit-to-cost model to select packages pre-installed in zygotes, and builds a tree cache to ensure that the forked zygote container does not import any additional packages other than the private ones the handler specifies. The C/R (Checkpoint/Restore) [7, 31, 39] supporting the VM snapshotting [10, 28, 29, 41, 54] captures the state of a running instance as a checkpoint, and then restores it once cold startup. Observing that most functions only access a small fraction of the files and mem-

ory loaded in the initialization stage, Catalyzer [29] and Replayable Execution [55] extend the C/R mechanism to achieve a faster on-demand recovery and paging when start containers. REAP [52] identifies the guest-side page when loading a VM snapshot and records the metadata during the record phase. Then, for subsequent invocations, REAP proactively prefetches and load the recorded pages into the guest memory for faster and higher-concurrency startup.

The above studies reduce the startup and recovery phases to partially improve the capability of higher-concurrency startup. From a different angle, RunD holistically focuses on prominent bottlenecks through a guest-to-host investigation when start secure containers with high-concurrency. We also proposes a lightweight serverless runtime that production-verfied in practice.

7 Conclusion

In serverless computing, the lightweight and short-term functions leads to the requirement of high-density container deployment and high-concurrency container startup. This work dives into the bottlenecks from the entire software stack and proposes RunD, a lightweight secure container runtime for serverless through a holistic guest-to-host solution. The evaluation results and in-production usage prove the efficiency of RunD to launch 200 secure containers in one second, and deploy over 2,500 secure containers per node. RunD is used in Alibaba production serverless platform, and shows good performance in terms of high-density deployment and high-concurrency startup.

Acknowledgment

We would thank Tianlong Wu, Haoran Yang, Xing Di, Xianbin Tang, Tao Ma, Jiang Liu, Zhiyuan Hou, Lei Wang, Zheng Liu, Gang Deng and Huaixin Chang from Alibaba Group for their contributions to this work. We also thank Jon Howell and our anonymous reviewers, for their helpful comments and suggestions.

This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61872240), and Shanghai international science and technology collaboration project (21510713600). Quan Chen and Minyi Guo are the corresponding authors.

References

[1] gvisor: Protecting gke and serverless users in the real world. cloud.google.com/blog/products/containers-kubernetes/how-gvisor-protects-google-cloud-services..., 2020.

[2] Alibaba function compute. https://alibabacloud.com/product/function-compute, 2021.

[3] Alpine linux. https://www.alpinelinux.org, 2021.

[4] Aws lambda. https://aws.amazon.com/lambda/, 2021.

[5] Aws lambda: The state of serverless. https://www.datadoghq.com/state-of-serverless-2020/, 2021.

[6] Container runtime interface (cri) - a plugin interface which enables kubelet to use a wide variety of container runtimes. https://github.com/kubernetes/cri-api, 2021.

[7] Criu: A utility to checkpoint/restore linux tasks in userspace. https://github.com/checkpoint-restore/criu, 2021.

[8] Device-mapper. http://www.sourceware.org/dm/, 2021.

[9] Execute mode in fission. https://docs.fission.io/docs/usage/executor/, 2021.

[10] Firecracker snapshotting. https://github.com/firecracker-microvm/firecracker/blob/master/docs/snapshotting/snapshot-support.md, 2021.

[11] Google cloud functions. https://cloud.google.com/functions, 2021.

[12] Lightweight pause container. https://groups.google.com/g/kubernetes-users/c/jVjv0QK4b_o, 2021.

[13] Microsoft azure functions. https://azure.microsoft.com/en-us/services/functions, 2021.

[14] Microsoft azure functions traces. https://github.com/Azure/AzurePublicDataset, 2021.

[15] Qemu. https://www.qemu.org, 2021.

[16] virtio-fs. https://virtio-fs.gitlab.io, 2021.

[17] What is vm templating and how to enable it. https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/what-is-vm-templating-and-how-do-I-use-it.md, 2021.

[18] Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.

[19] Kata containers - open source container runtime software. https://katacontainers.io/, 2022.

[20] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 419–434. USENIX Association, 2020.

[21] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: towards high-performance serverless computing. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 923–935. USENIX Association, 2018.

[22] S. Barlev, Z. Basil, S. Kohanim, R. Peleg, S. Regev, and Alexandra Shulman-Peleg. Secure yet usable: Protecting servers and linux containers. *IBM J. Res. Dev.*, 60(4):12, 2016.

[23] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46. USENIX, 2005.

[24] Guillaume Bonfante, Jean-Yves Marion, and Daniel Reynaud-Plantey. A computability perspective on self-modifying programs. In Dang Van Hung and Padmanabhan Krishnan, editors, *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 231–239. IEEE Computer Society, 2009.

[25] Marcus Botacin, Marco Antonio Zanata Alves, and André Grégio. The self modifying code (smc)-aware processor (SAP): a security look on architectural impact and support. *J. Comput. Virol. Hacking Tech.*, 16(3):185–196, 2020.

[26] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo N. Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco A. S. Netto, Adel Nadjaran Toosi, Maria Alejandra Rodriguez, Ignacio Martín Llorente, Sabrina De Capitani di Vimercati, Pierangela Samarati, Dejan S. Milojicic, Carlos A. Varela, Rami Bahsoon, Marcos Dias de Assunção, Omer Rana, Wanlei Zhou, Hai Jin, Wolfgang Gentzsch, Albert Y. Zomaya, and Haiying Shen. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM Comput. Surv.*, 51(5):105:1–105:38, 2019.

[27] Quan Chen, Shuai Xue, Shang Zhao, Shanpei Chen, Yihao Wu, Yu Xu, Zhuo Song, Tao Ma, Yong Yang, and Minyi Guo. Alita: comprehensive performance isolation through bias resource management for public clouds. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 32. IEEE/ACM, 2020.

[28] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In Amin Vahdat and David Wetherall, editors, *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. USENIX, 2005.

[29] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 467–481. ACM, 2020.

[30] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In Michael B. Jones, editor, *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 251–266. ACM, 1995.

[31] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. *login Usenix Mag.*, 41(4), 2016.

[32] Ricardo Koller and Dan Williams. An ounce of prevention is worth a pound of cure: Ahead-of-time preparation for safe high-level container interfaces. In Daniel Peek and Gala Yadgar, editors, *11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, Renton, WA, USA, July 8-9, 2019*. USENIX Association, 2019.

[33] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 11:1–11:15. ACM, 2020.

[34] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Comput. Surv.*, dec 2021. Just Accepted.

[35] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 782–796. ACM, 2022.

[36] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 461–472. ACM, 2013.

[37] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 218–233. ACM, 2017.

[38] Massimiliano Mattetti, Alexandra Shulman-Peleg, Yair Allouche, Antonio Corradi, Shlomi Dolev, and Luca Foschini. Securing the infrastructure and the workloads of linux containers. In *2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015*, pages 559–567. IEEE, 2015.

[39] M. Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino, editors, *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 405–410. IEEE Computer Society, 2017.

[40] Anup Mohan, Harshad Sane, Kshitij Doshi, and Saikrishna Edupuganti. Agile cold starts for scalable serverless. In Christina Delimitrou and Dan R. K. Ports, editors, *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*. USENIX Association, 2019.

[41] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 391–394. USENIX, 2005.

[42] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: rapid task provisioning with serverless-optimized containers. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 57–70. USENIX Association, 2018.

[43] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In Jennifer B. Sartor, Mayur Naik, and Chris Rossbach, editors, *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019, Providence, RI, USA, April 14, 2019*, pages 59–73. ACM, 2019.

[44] Pu Pang, Quan Chen, Deze Zeng, and Minyi Guo. Adaptive preference-aware co-location for improving resource utilization of power constrained datacenters. *IEEE Trans. Parallel Distributed Syst.*, 32(2):441–456, 2021.

[45] Rob Pike, David L. Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. *Comput. Syst.*, 8(2):221–254, 1995.

[46] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[47] Florian Schmidt. uniprof: A unikernel stack profiler. In *Posters and Demos Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 31–33. ACM, 2017.

[48] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pages 1063–1075. ACM, 2019.

[49] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 205–218. USENIX Association, 2020.

[50] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Wenji Li, Raju Rangaswami, and Ming Zhao. Evaluating docker storage performance: from workloads to graph drivers. *Clust. Comput.*, 22(4):1159–1172, 2019.

[51] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, NagaPramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. In search of the ideal storage configuration for docker containers. In *2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017*, pages 199–206. IEEE Computer Society, 2017.

[52] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 559–572. ACM, 2021.

[53] William Viktorsson, Cristian Klein, and Johan Tordsson. Security-performance trade-offs of kubernetes container runtimes. In *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2020, Nice, France, November 17-19, 2020*, pages 1–4. IEEE, 2020.

[54] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 148–162. ACM, 2005.

[55] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 39:1–39:16. ACM, 2019.

[56] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. Peeking behind the curtains of serverless platforms. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 133–146. USENIX Association, 2018.

[57] Andrew Whitaker, Marianne Shaw, and Steven Gribble. Denali: Lightweight virtual machines for distributed and networked applications. 03 2002.

[58] Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma. Adaptive function launching acceleration in serverless computing platforms. In *25th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2019, Tianjin, China, December 4-6, 2019*, pages 9–16. IEEE, 2019.

[59] Shuai Xue, Shang Zhao, Quan Chen, Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou, Keqiang Niu, Sijie Sun, and Minyi Guo. Spool: Reliable virtualized nvme storage pool in public cloud infrastructure. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 97–110. USENIX Association, 2020.

[60] Yang Zhan, Alexander Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. How to copy files. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 75–89. USENIX Association, 2020.

[61] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. Understanding, predicting and scheduling serverless workloads under partial interference. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, pages 22:1–22:15. ACM, 2021.

A Artifact Appendix

A.1 Abstract

We choose Kata containers and its three configurations *kata-qemu*, *kata-FC*, *kata-template* as baselines for comparison with RunD. For measuring the startup latency, we use the

crictl command to start pod sandboxes and measure the time between the first *crictl runp* invocation and the last ready pod sandbox. For measuring the memory footprint, we use the *smem* command and the PSS column of its output. All the tests are run on a machine with 104 vCPUs and 384GB of memory running CentOS7.

A.2 Artifact Check-list (Meta-information)

- **Run-time environment:** Alibaba ECS instance;
- **Hardware:** Intel Xeon(Cascade Lake) Platinum 8269CY, CPU and Memory: 104 cores and 384GiB, Storage: Two ESSDs (100GB + 500GB);
- **Software:** Aliyun Cloud OS 2, with Linux kernel 4.19.91, Kata container 1.12.1 and 2.2.3, containerd 1.3.10, smem 1.4;
- **Metrics:** average latency and average memory footprint;
- **Time is needed to complete experiments:** 10 hours;
- **Available:** https://github.com/chengjiagan/RunD_ATC22
- **Code Licenses:** Apache-2.0 license

A.3 How to Access and Installation

Github Link: https://github.com/chengjiagan/RunD_ATC22. Then you should follow the *README* instructions to get installation.

A.4 Experiment Workflow

A.4.1 High-concurrency Experiment (Section 5.2)

Scripts are provided to run the high-concurrency test for kata-qemu, kata-fc and kata-template. To run high-concurrency tests:

```
$ ./script/time_kata_test.sh
$ ./script/time_katafc_test.sh
$ ./script/time_katatemplate_test.sh
```

They may take several hours to finish. Some concurrency tests can be removed by removing the corresponding concurrency setting in file *time_test.conf* to shorten the time. The scripts will create a directory (e.g., named like *time_kata_05120948*) to store the logs.

We provide python scripts to analyze logs from the tests:

```
$ python3 data/time.py
$ python3 data/cpu.py
```

The python script will create two .csv files in the result directory: *time.csv* and *cpu.csv*. Each line in the csv file indicates the average cold-start latency and cpu time of a container runtime.

A.4.2 High-density Experiment (Section 5.3)

Scripts are provided to run the high-density test for kata-qemu, kata-fc and kata-template. To run high-density tests:

```
$ ./script/mem_kata_test.sh
$ ./script/mem_katafc_test.sh
$ ./script/mem_katatemplate_test.sh
```

Density and memory capacity of containers in the tests can be changed in the file *mem_test.conf*. The scripts will create a directory named like *mem_kata_05120948* to store the logs.

We provide a python script to analyze logs from the tests:

```
$ python3 data/mem.py
```

The python script will create a csv file for each runtime, named like `mem_kata.csv`, containing the average memory consumption of containers with different memory capacity in different density.

A.4.3 Density Impact on Concurrency (Section 5.4)

Scripts are provided to run the high-density test for kata-qemu, kata-fc and kata-template. To run tests:

```
$ ./script/density_kata_test.sh
$ ./script/density_katafc_test.sh
$ ./script/density_katatemplate_test.sh
```

The background density and the concurrency of the tests can be changed in the file *density_test.conf*. The scripts will create a directory (e.g., named like *density_kata_05120948*) to store the logs.

We provide a python script to analyze logs from the tests:

```
$ python3 data/density.py
```

The python script will create a csv file for each runtime, named like *density_kata.csv*, containing the average cold-start latency under different background densities and concurrencies.

A.5 Expected Results and Notes

The expect results are all stored in *ae_data* directory. Considering that some related binary packages are tightly integrated with our internal system, we provide a screencast *ATC_RunD_AE.mp4* of the tool along with the results. You can also find RunD-related performance and execution logs in our artifact..