# *RainbowCake*: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing

Hanfei Yu
Louisiana State University

Rohan Basu Roy
Northeastern University

Christian Fontenot
Louisiana State University

Devesh Tiwari
Northeastern University

Jian Li
Stony Brook University

Hong Zhang
University of Waterloo

Hao Wang
Louisiana State University

Seung-Jong Park
Missouri University of Science and Technology

## Abstract

Serverless computing has grown rapidly as a new cloud computing paradigm that promises ease-of-management, cost-efficiency, and auto-scaling by shipping functions via self-contained virtualized containers. Unfortunately, serverless computing suffers from severe cold-start problems—starting containers incurs non-trivial latency. Full container caching is widely applied to mitigate cold-starts, yet has recently been outperformed by two lines of research: partial container caching and container sharing. However, either partial container caching or container sharing techniques exhibit their drawbacks. Partial container caching effectively deals with burstiness while leaving cold-start mitigation halfway; container sharing reduces cold-starts by enabling containers to serve multiple functions while suffering from excessive memory waste due to over-packed containers.

This paper proposes *RainbowCake*, a layer-wise container pre-warming and keep-alive technique that effectively mitigates cold-starts with sharing awareness at minimal waste of memory. With structured container layers and sharing-aware modeling, *RainbowCake* is robust and tolerant to invocation bursts. We seize the opportunity of container sharing behind the startup process of standard container techniques. *RainbowCake* breaks the container startup process of a container into three stages and manages different container layers individually. We develop a sharing-aware algorithm that makes event-driven layer-wise caching decisions in real-time. Experiments on OpenWhisk clusters with real-world workloads show that *RainbowCake* reduces 68% function startup latency and 77% memory waste compared to state-of-the-art solutions.

*CCS Concepts:* • **Computer systems organization** → **Cloud computing**.

*Keywords:* Serverless Computing, Container Caching, Container Sharing, Cold-start

**ACM Reference Format:**
Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. *RainbowCake*: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3617232.3624871

## 1 Introduction

Serverless computing, as a new programming paradigm, has revolutionized how applications utilize cloud resources. Various application domains, such as web services, online video processing, data analytics, scientific computing, and machine learning [3, 4, 14, 24, 30, 33, 46, 54], have embraced serverless computing model as this model simplifies how users deploy applications. Serverless platforms relieve users from heavy daily operations (*e.g.*, infrastructure maintenance and resource provisioning) with transparent auto-scaling. Users only need to upload their code onto serverless platforms as *functions*[1] and invoke the functions, leaving everything else to the service provider.

[1]In this paper, a function refers to an executable code package deployed on a serverless platform, and a function invocation is a running instance of the package. A function can be invoked multiple times over time.

Serverless functions are latency-sensitive due to their short execution time—often in the range of seconds [53]—leaving limited time for platforms to prepare a container[2], which takes over hundreds of milliseconds, severely inflating function response time, known as the notorious *cold-start* problem. Fig. 1 shows the landscape of recent studies proposed to mitigate cold-starts in serverless computing, including new container techniques [1, 27], checkpointing [5, 21, 55], memory deduplication [49], and caching (*i.e.*, pre-warming [12, 29, 48, 53] and keep-alive [25, 45, 48, 53]).

However, serverless workloads are highly volatile, bursty, and hard-to-predict—recent research indicates that over 50% of functions on Microsoft Azure Function witness significantly varying invocation patterns [53]. The *full*-container caching has been widely applied as an effective solution addressing the burstiness in serverless computing [12, 25, 29, 45, 48, 53]. A *fully* initialized container is also fully specialized, only serving requests of one specific function but occupying hundreds of MBs in memory. Thus, any mispredictions may lead to frequent cold-starts and vast memory waste [13].

A few recent studies attempt to accommodate the dilemma by seeking fine-grained trade-offs from two main perspectives: 1) *partial container caching* approaches break a container into fragments [39, 42] or layers [13] and caches partial containers to reduce memory waste while accelerating container initialization; 2) *container sharing* schemes [2, 23, 37, 40, 43, 50] enable an idle container to serve multiple functions by packing their common dependencies (or libraries) into one monolithic container image, thus tolerating potential mispredictions.

Intuitively, given the same memory budget, partial container caching increases the number of containers in memory, thus raising the hit rate of partially initialized containers. Instead, container sharing decreases the number of containers in memory due to their increased size (by packing more dependencies) but generalize a container to be hit by different functions, which raises the hit rate of the "over-packed" containers. However, both partial caching and container sharing solutions fail to strike a *fundamental* balance between cold-start mitigation and memory wasting, thus leading to either halfway cold-start mitigation or excessive memory waste.

We argue that partial container caching and sharing must be synergized to mitigate cold-starts with minimal memory waste. Fig. 1 describes *RainbowCake*'s position in the design space of cold-start mitigation solutions. *RainbowCake* is the first serverless cold-start mitigation technique that enjoys the merits from both sides, *i.e.*, joint partial container caching and sharing. *RainbowCake* enables layer-wise container caching and sharing by carefully exploiting different

[2]We use the term "container" to denote general virtual environments that execute function invocations in serverless computing, such as Docker containers and Firecracker MicroVMs.

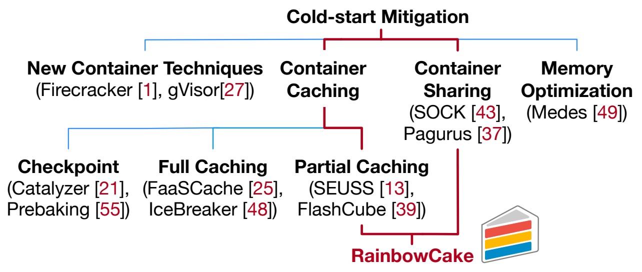
Cold-start Mitigation
New Container Techniques (Firecracker [1], gVisor[27])
Container Caching
Container Sharing (SOCK [43], Pagurus [37])
Memory Optimization (Medes [49])
Checkpoint (Catalyzer [21], Prebaking [55])
Full Caching (FaaSCache [25], IceBreaker [48])
Partial Caching (SEUSS [13], FlashCube [39])
RainbowCake

**Figure 1.** Design space for cold-start mitigation.

layers' generalities: *Lower-level container layers are lighter and can be shared by more functions. Higher-level layers save more start-up latency but are heavier and more specialized.* Based on this insight, *RainbowCake* leverages invocations' historical information to customize pre-warm and keep-alive strategies tailored to every function. Overall, we make the following key contributions:

- We carefully examine layer-wise container initialization process and identify three layers that naturally decouple the initialization process without breaking any system dependencies that enable *RainbowCake* with layer-wise partial container *caching* and *sharing*.
- We propose a layer-wise sharing-aware algorithm to mitigate cold-starts with minimal memory wasting by proactively pre-warm and adaptively keep-alive layer-wise containers, achieving a high tolerance to burstiness.
- We implement *RainbowCake* in OpenWhisk and deploy it on a real cluster. Extensive experiments with industrial traces and real-world applications show that *RainbowCake* reduces 68% function startup latency and 77% container memory waste compared to state-of-the-art solutions.

## 2 Background and Motivation

We describe the cold-start problem in serverless computing (§2.1), then briefly introduce existing cold-start mitigation techniques with a focus on partial container caching and sharing (§2.2), and motivate the necessity of a joint design (§2.3).

### 2.1 Cold-starts in Serverless Computing

Fig. 2(a) illustrates the varying cold-start latencies of 20 realistic functions executed on an OpenWhisk cluster with Docker containers, which inflate the end-to-end function response time significantly [53, 57]. Similar to existing partial container caching works [13], we carefully identify the following three key stages in serverless function cold-starts:

**Stage #1: Environment setup**. Upon receiving a function invocation, the serverless computing platform prepares the infrastructural environment (*e.g.*, selecting a worker node and checking the container pool) for container initialization and execution. A container proxy is created to provide the

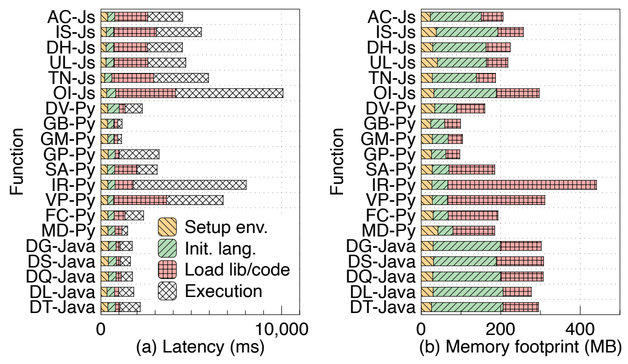

**Figure 2.** Cold-start latency and memory footprint breakdown of three stages for 20 realistic functions of Node.js, Python, and Java (details in §7.1).

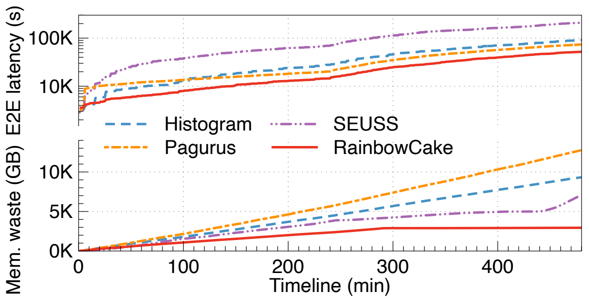

**Figure 3.** Timelines of running the 20-function workload driven by eight-hour Azure Functions traces. Metrics include cumulative function end-to-end latency and cumulative memory waste.

necessary utilities for the container, such as logging, communication, and container health checking.

**Stage #2: Language runtime initialization**. After the container proxy is ready, the platform pulls an image into the container for the invocation's corresponding language runtime and related built-in libraries to initialize the runtime. For example, the invocation to IR-Py will create a container based on an image with Python runtime.

**Stage #3: User deployment package loading**. After the container is created, the platform then loads the user deployment code package into the container, including user code, execution tools, libraries, and other dependencies. Fig. 2(b) shows that the memory consumption varies between different languages and user deployment packages.

The three stages of cold-starts and their varied latency and memory footprints imply the existence of a more fundamental trade-off between cold-start latency and memory consumption, motivating the idea of *RainbowCake* that *partially* pre-warms & keeps alive containers initialized to one of the three stages at runtime.

## 2.2 Recent Trends in Cold-start Mitigation

Existing pre-warming & keep-alive strategies [12, 25, 29, 45, 48, 53] make coarse-grained decisions, *i.e.*, either reserving a full-size container or completely terminating the container, can hardly find the optima between cold-starts and resource wasting. Choosing either decision can lead to severe problems—keeping a full-size container alive wastes hundreds of MBs of memory if no invocations to serve, while terminating the container leads to potential performance degradation due to cold-starts. Recently, researchers have attempted to tackle the coarse-grained trade-off in mitigating cold-starts in two directions: *partial container caching* [13, 39, 42] and *container sharing* [37, 43].

**Partial container caching** solutions resort to finer-grained trade-offs by breaking a container into fragments, such as network connectors [42] and namespace constructors [39]; or layered structures based on the common three stages [13] in §2.1. The key idea behind partial container caching is to keep containers with only partial fragments or layers in memory, so that memory cost can be effectively reduced while cold-starts are mitigated with partial containers.

**Container sharing** aims to avoid the trade-off by recycling idle containers from one function to help other functions. Containers of serverless functions are typically private—every container is assigned to a specific function and incompatible with others. Full container caching techniques inevitably accumulate (private) idle containers due to imperfect pre-warming or keep-alive policies. Sharing approaches utilize those idle containers to serve other functions (selected via tree cache [43] or weighted sampling [37]), thus further reducing cold-starts. Commercial serverless platforms' limited language versions (*e.g.*, AWS Lambda supports at most four Python versions [9]) also make container sharing practical.

However, either partial container caching or container sharing techniques have non-trivial downsides. Next, we use a real-world experiment to demonstrate a co-design by jointly applying the two methods to achieve an optimum.

## 2.3 Layer-wise Caching and Sharing

In the prototype of *RainbowCake* on OpenWhisk [6], we identify three types of containers according to their initialized layers in bottom-up order. The three container types are aligned with existing partial container caching solutions (*e.g.*, the three initialization paths in SEUSS [13]):

- A **`Bare` container** has only initialized the infrastructural environment and utilities for containers (*e.g.*, establishing network connections and logs). It has a low memory consumption and is compatible with any functions but has to install a language runtime and load user code before executing an invocation.
- A **`Lang` container** has initialized a container with a language runtime (*e.g.*, Python) based on the `Bare` container. It is compatible with functions of the same language (*e.g.*, DV-Py and IR-Py). Compared to a `Bare`

container, a Lang container consumes more memory but takes less time to prepare for invocation execution.

- A **User container**, also known as a full-size container, has initialized user library/codebase (*e.g.*, IR-Py) based on the Lang container. It is instantly ready to execute an invocation but only compatible with one function (*e.g.*, IR-Py).

To motivate the need of *RainbowCake*, we run a serverless workload [17, 18, 52] driven by 8-hour industrial invocation traces [53]. Detailed experimental setup can be viewed in §7.1. Fig. 3 shows the cumulative function end-to-end latency and cumulative memory cost, respectively. We implement Histogram [53], SEUSS [13], and Pagurus [37] in OpenWhisk as baselines to represent full container caching, partial container caching, and container sharing, respectively. Compared to full container caching (Histogram) and container sharing (Pagurus), partial container caching (SEUSS) significantly reduces the memory cost for keeping containers. However, partial warm-starts delivered by partial container caching fail to match the latency reduction of complete warm-starts. Though container sharing reduces cold-starts and accelerates function startup, shareable full-size containers are overweight due to over-packed, resulting in excessive memory waste.

With a joint design of partial container caching and sharing, our proposed *RainbowCake* can achieve both a low function end-to-end latency and minimal memory cost. *RainbowCake* carefully makes pre-warm & keep-alive decisions on caching proportions of containers to different layers at runtime with cross-function sharing awareness.

## 3 *RainbowCake* Overview

### 3.1 Objectives and Challenges

We carefully design *RainbowCake* to achieve three goals:

**Mitigating cold-starts with minimal resource wasting**. *RainbowCake* is designed to proactively pre-warm and keep-alive containers with fine-grained layer-wise caching decisions. With a joint design, *RainbowCake* should enable function invocations to reuse and share containers with different layers. **Tolerance to burstiness and mispredictions**. Serverless computing workloads are typically event-driven and bursty, making it hard to accurately predict the arrivals of invocation bursts. *RainbowCake* should effectively mitigate cold-starts and maintain stable resource wasting when dealing with such burstiness and potential mispredictions.

**Lightweight and high scalability**. Serverless computing workloads are typically latency-sensitive (*e.g.*, sub-second response time) and involve large-scale concurrent invocations. *RainbowCake* should be able to effectively mitigate cold-starts of high concurrent invocations without introducing non-negligible overhead.

To achieve the above objectives, we need to answer the following challenging questions:

**How to find the optimum of the trade-off between cold-start and resource wasting?** Introducing fine-grained container layers (*i.e.*, Bare layer, Lang layer, and User layer) extensively complexifies the decision space of trade-offs between cold-start latency and resource wasting. Besides, the diversity of language runtimes and user deployment packages as well as their varying cold-start overheads further complicate the solution search space.

**How to tune *RainbowCake*'s sensitivity and tolerance of burstiness and mispredictions?** Ideally, perfect predictions enable pre-warming & keep-alive methodologies to achieve 100% hit rate of full-size containers—zero cold-starts. In reality, with the existence of mispredictions, how *RainbowCake* adapts to bursty invocations by tweaking the containers' compatibility to function types is a non-trivial decision.

**How to be compatible with various container software stacks of different serverless platforms?** Serverless computing has a rich and diverse software stack, such as Firecracker [1] and gVisor [27], involving different virtualization techniques, programming frameworks, dependencies, and languages. *RainbowCake*'s layer-wise pre-warming and keep-alive must be sufficiently general to support effective cold-start mitigation for the serverless computing eco-system.

### 3.2 Architecture

*RainbowCake* has two key components: the *History Recorder* and the *Container Pool*. The *History Recorder* keeps observing invocation request arrivals and captures the invocation patterns by fitting sharing-aware distribution models with the collected invocation arrival records (§5.1). The *Container Pool* maintains containers of different layers and executes event-driven pre-warming and keep-alive operations according to the estimation of upcoming invocations (§5.2).

### 3.3 Workflow

Fig. 4 presents an example of our workflow by comparing *RainbowCake* with existing keep-alive approaches. Full container caching approaches keep full-size containers (with User layer) alive before termination. The containers stay idle unless reused by invocations from the same function, resulting in container resource wasting. In the top of Fig. 4, Function A invocation arrives, executes with a container, and then starts to keep alive. During the keep-alive period of A's container, Function B invocation arrives but cannot reuse A's container due to incompatible User-layer, which has to start a new container for execution with cold-start. Container sharing solutions pack multiple libraries in one container with a heavy User layer, yet may still suffer from cold-starts. In the middle of Fig. 4, Function B invocation can reuse the idle container, but C invocation is cold-started due to incompatible User layers.

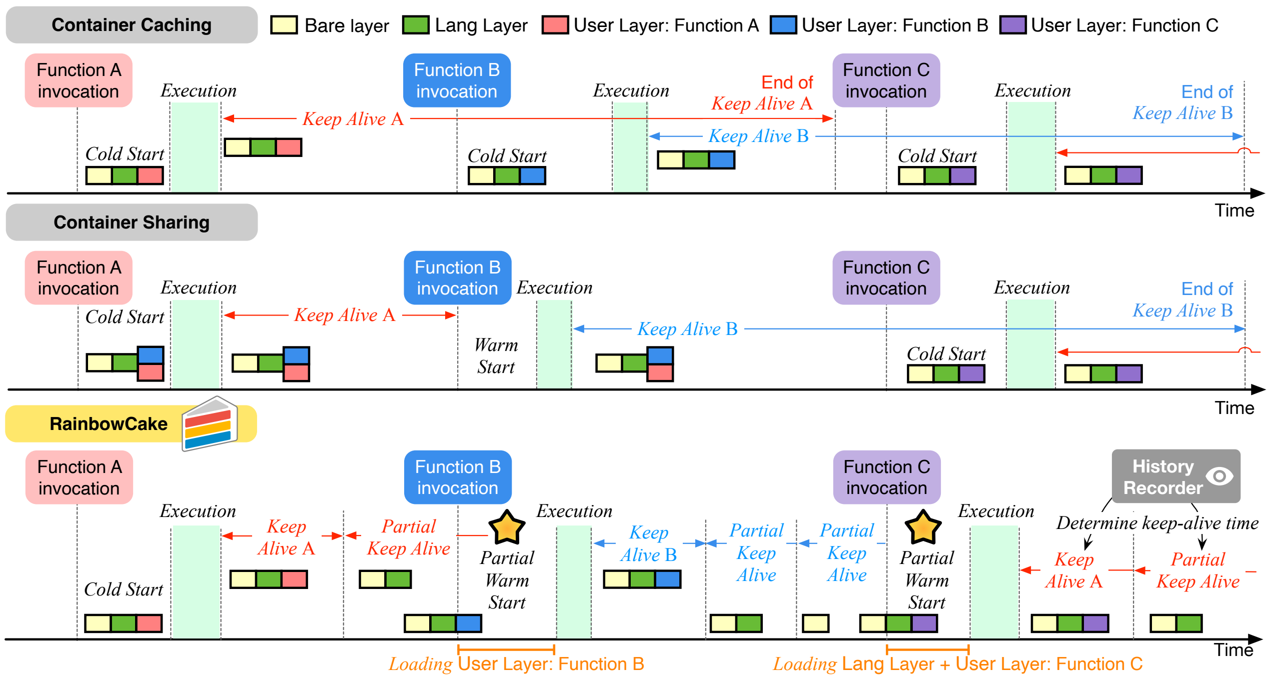

**Figure 4.** *RainbowCake*'s workflow.

In contrast to existing container caching and sharing approaches, *RainbowCake* enables idle containers *partial* keep-alive ability for safe and cross-function reuse. In the bottom of Fig. 4, we keep A's User container alive for a while, then peel off the User layer of A, which downgrades to a Lang container. Function B invocation can now reuse the Lang container since they have the same language runtime. B invocation reuses the Lang container by installing its own User layer inside it, which can be considered as a *partial warm-start* that saves the startup time of installing Lang and Bare layers. Similarly, we peel off the User and Lang layers of B's container during keep-alive, and C invocation can still reuse the Bare container of B with a partial warm-start to reduce startup latency.

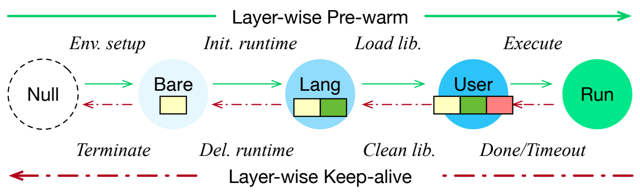

**Figure 5.** Life cycle and state transition of a container in *RainbowCake*.

## 4 Problem Formulation

### 4.1 System Model

We consider a workload with a set of functions served by a serverless platform. Each function has a specific user deployment package and a language runtime, and each function invocation takes a container with suitable language runtime and deployment packages to execute. Let b, l, u denote Bare container, Lang container, User container, respectively. We assume the platform owns a limited amount of resources (*e.g.*, memory) for keeping multi-layered containers. A newly-arrived invocation either hits an available container in one of the three types (Bare, Lang, and User) with warm-start or triggers the initialization of a new container with cold-start.

Bare containers can serve any invocations from arbitrary functions, Lang containers can be shared by any invocations of the same language runtime (*e.g.*, an Lang containers with Python can be reused among GM, GB, VP, IR, and DV invocations in Fig. 2), and User containers can only be reused by invocations launched within the same function. The platform offers pre-warming and keep-alive services for every function. Concretely, the platform pre-warms containers to one of the three types to serve future function invocations. The containers start their keep-alive period immediately after either being pre-warmed or completing execution. Hence, a running container always belongs to one of the three types (*i.e.*, Bare, Lang, and User). Fig. 5 depicts the state transition among different types and the life cycle of multi-layered containers. A container upgrades to the next type when installing the corresponding layer and downgrades to the previous type after its current time window expires.

### 4.2 Cost Metrics

We use two metrics to measure the performance of a serverless platform addressing cold-starts for a workload: 1) *startup overhead* of a function invocation refers to the time starting from preparing a container until actual execution, and

2) *wasted resource* of a container refers to the resources occupied during the idle phase without serving invocations. We consider each metric as a cost that the platform takes to serve a workload.

Let $C_{startup}^{(k)}$ and $C_{memory}^{(k)}$ denote the total startup cost and total resource waste cost for handling all invocations of a function at type $k \in \{\mathsf{b}, \mathsf{l}, \mathsf{u}\}$, respectively. Across all three types of layers, the total startup cost $C_{startup}$ and total memory waste cost $C_{memory}$ are summed by

$$C_{startup} := \sum_{k \in \{\mathsf{b},\mathsf{l},\mathsf{u}\}} C_{startup}^{(k)},\ C_{memory} := \sum_{k \in \{\mathsf{b},\mathsf{l},\mathsf{u}\}} C_{memory}^{(k)}.$$

Thus, the unified cost $C$ consisting of the startup cost and the resource waste cost is given by

$$C := \alpha \times C_{startup} + (1 - \alpha) \times C_{memory}, \tag{1}$$

where parameter $\alpha \in (0, 1)$ serves as a tunable knob that the platform can trade off between two types of costs.

### 4.3 Objective and Complexity

To serve future function invocations, the platform keeps a certain number of containers by pre-warming and keep-alive for each type $k \in \{\mathsf{b}, \mathsf{l}, \mathsf{u}\}$. In real-time, we make two kinds of decisions for each type $k \in \{\mathsf{b}, \mathsf{l}, \mathsf{u}\}$: pre-warming and keeping containers alive. The objective is to minimize the unified cost in Eq. 1 over the total time span of a workload, where finding the optimal set of decisions can be naturally formulated as an Integer Linear Programming (ILP) problem with NP-completeness. However, searching for the optimum in large-scale serverless platforms with sub-second latency requirements and online invocation arrivals is extremely challenging. Therefore, we opt for an efficient event-driven design of *RainbowCake* to enable fine-grained layer-wise pre-warming and keep-alive decisions.

## 5 *RainbowCake* Design

### 5.1 Sharing-aware Invocation Modeling

We leverage invocation history to model the sharing-aware arrival distributions of hits on each container type.

**Function-specific modeling.** To capture the latest invocation patterns, we first fit the invocation distribution of function $f$ using a sliding window on the latest $n$ invocations (*e.g.*, six invocations). We record the arrival timestamp $j'$ of the stalest invocation in the window to calculate the rate parameter of the distribution $\lambda_f = \frac{n}{j-j'}$, where $j$ is the current timestamp. For each function, we model a *Poisson distribution* with random variable $X_f \sim Poisson(\lambda_f)$. Poisson distribution is commonly applied to describe request arrival patterns [22, 26, 34, 56].

**Sharing-aware estimation.** We employ *compound Poisson distribution* for the sharing-aware invocation modeling. Given the invocation history of a set of functions $F$, each function $f \in F$ is modeled by a function-specific Poisson distribution with a random variable $X_f \sim Poisson(\lambda_f)$. Let $F^{(k)}$ denote the set of functions equipping with $k$-type layer within $F$, where $k \in \{\mathsf{b}, \mathsf{l}, \mathsf{u}\}$. For each container type $k$, we compound the corresponding function-specific distributions to be a Poisson distribution $Y^{(k)}$:

**Algorithm 1:** *RainbowCake*'s Pre-warming

```
1  async def SchedulePrewarm(function_id, IAT):
2      Sleep(IAT) /* Wait until next request */
3      if Available(function_id) is False then
           /* Pre-warm if no warm ones */
4          PrewarmContainer(function_id, type=User)
5      else
           /* Skip if warm containers exist */
6          pass
7      return
8  while function invocation arrives do
9      function_id ← function.get_id()
10     next_IAT ← Poisson(function_id, type=User)
       /* Asynchronous execution */
11     SchedulePrewarm(function_id, next_IAT)
```

$$Y^{(k)} \sim Poisson(\lambda^{(k)}) = \sum_{f \in F^{(k)}} X_f, \tag{2}$$

where $\lambda^{(k)} = \sum_{f \in F^{(k)}} \lambda_f$. The compounded summand in Eq. 2 can be transformed to an equivalent *exponential distribution* with the following cumulative distribution function (CDF):

$$CDF(x; \lambda^{(k)}) = \begin{cases} 1 - e^{\lambda^{(k)} x} & x \geq 0, \\ 0 & x < 0, \end{cases} \tag{3}$$

Using the CDF in Eq. 3, we can estimate the probability of a given inter-arrival time (IAT) of the distribution, where IAT is defined as the time interval between two consecutive invocation arrivals. Inversely, given a quantile $p$, the IAT of type $k$ can be derived from the *quantile function* of $Exp(\lambda^{(k)})$:

$$IAT(k, p) = CDF^{-1}(p; \lambda^{(k)}) = \frac{-ln(1-p)}{\lambda^{(k)}},\ 0 \leq p < 1. \tag{4}$$

Thus, by specifying $p$, we can estimate an invocation arriving in at most $IAT(k, p)$ time. Intuitively, the quantile $p$ represents the confidence of predicting future invocations—the higher $p$, the longer *RainbowCake* tends to keep partial containers alive. *RainbowCake* further uses the estimated IATs to determine pre-warming and keep-alive Time-to-Live (TTL) decisions.

### 5.2 Layer-wise Pre-warming and Keep-alive

*RainbowCake* enforces TTL-based pre-warming and keep-alive strategies when provisioning fine-grained containers layer-wise. However, pre-warming and keep-alive decisions

**Algorithm 2:** *RainbowCake*'s Keep-alive

```
def ComputeTTL(container, IAT):
    $t \leftarrow$ container.get_startup_latency()
    $m \leftarrow$ container.get_memory_footprint()
    $\beta \leftarrow (\alpha \times t)/((1-\alpha) \times m)$ /* Equation 6 */
    return Min(IAT, $\beta$)
while container timeouts do
    function_id $\leftarrow$ container.get_function_id()
    layer $\leftarrow$ container.get_type()
    if layer is Bare then
        /* Bare containers timeout */
        container.kill()
    else
        /* User or Lang containers timeout */
        container.downgrade()
        layer $\leftarrow$ container.get_type()
        next_IAT $\leftarrow$ Poisson(function_id, layer)
        TTL $\leftarrow$ ComputeTTL(container, next_IAT)
        SetContainerTimeout(container, TTL)
```

are highly correlated with each other. Finding an optimal solution is impractical in serverless platforms with sub-second latency requirements (§4).

Therefore, we propose an *event-driven* heuristic algorithm to dynamically adjust the length of pre-warming and keep-alive TTLs in real-time. The event-driven design is aligned with the serverless computing nature, where serverless function invocations are highly volatile, bursty, and usually triggered by various user events. The algorithm utilizes sharing-aware distributions modeled from historical invocation information to adapt the length of TTLs. *RainbowCake*'s algorithm consists of two parts: *pre-warming* and *keep-alive*.

**Pre-warming**. Upon the latest invocation arrives, *RainbowCake*'s container pool queries the function history to estimate the IAT in Eq. 4. Then, the pool schedules a pre-warm event after the IAT time to pre-warm a User container for the function. When the event is scheduled, the container pool first checks whether any idle User containers of the function exist. If any, we ignore the pre-warming schedule since an incoming invocation can directly reuse idle User containers in the pool. Otherwise, we start pre-warming a User container for the function. Pre-warmed containers proceed to the keep-alive period immediately after startup, which can further downgrade and be reused by other invocations.

**Keep-alive**. Every idle container needs to uninstall three layers sequentially in a keep-alive period: User, Lang, and Bare. Before transitioning to the next type, a container notifies the container pool that stores the invocation distribution models. The pool uses the corresponding distribution's quantile function to estimate the IAT of invocations in the next type. We define $t^{(k)}$ as the startup latency of installing the dependencies of the corresponding layer, and define $m^{(k)}$ as the memory consumption of an idle container in type $k \in \{\mathtt{b}, \mathtt{l}, \mathtt{u}\}$. The startup cost $C^{(k)}_{startup}$ in Eq. 1 is measured by accumulating startup latency $t^{(k)}$. Since CPU allocation can be adjusted using CPU sharing techniques without terminating the container (*e.g.*, cpu-shares in Linux cgroups [31]), we measure the product of memory occupation $m^{(k)}$ and idle time to account for resource waste cost $C^{(k)}_{memory}$ in Eq. 1 of an idle container. The startup latency and memory footprint of an idle container for a function is typically constant, thus we use the average startup latency $\bar{t}^{(k)}$ and the average memory occupation $\bar{m}^{(k)}$:

$$\bar{t}^k = \frac{\sum_{i=1}^n t_i^{(k)}}{n}, \ \bar{m}^k = \frac{\sum_{i=1}^n m_i^{(k)}}{n}, \tag{5}$$

over the sliding window of $n$ invocations of the function for type $k$. We compute an upper bound $\beta^{(k)}$ for the predicted IAT of $k$ by assuming $\alpha \times \bar{t}^{(k)} = (1-\alpha) \times \bar{m}^{(k)} \beta^{(k)}$ in Eq. 1:

$$\beta^{(k)} := \frac{\alpha \bar{t}^{(k)}}{(1-\alpha)\bar{m}^{(k)}}, \ k \in \{\mathtt{b}, \mathtt{l}, \mathtt{u}\}. \tag{6}$$

The IAT upper bound $\beta$ in Eq. 6 dictates the maximum duration for a container to stay idle, thus constraining a container's memory waste cost cannot exceed its startup cost of the function. The TTL length for function $n$ keep-alive decisions with quantile $p$ at $k$ is given by

$$TTL(n,k,p) = \min\{IAT(k,p), \ \beta^{(k)}\}, \ k \in \{\mathtt{b}, \mathtt{l}, \mathtt{u}\}. \tag{7}$$

Algorithms 1 and 2 summarize *RainbowCake*'s event-driven pre-warming and keep-alive strategies. We design *RainbowCake* to make layer-wise pre-warming and keep-alive decisions driven by invocations. The event-driven design efficiently shrinks the decision space, which has immense exponential complexity analyzed in §4. We make decisions of pre-warming and keep-alive separately, where invocation arrivals trigger *RainbowCake* to make pre-warming decisions and idle container timeouts trigger keep-alive decisions. Both pre-warming and keep-alive require *RainbowCake* to perform a constant complexity of operations as described in §5.2. Hence, *RainbowCake*'s pre-warming and keep-alive decisions incur negligible operational overhead compared to the original decision space.

## 5.3 Security

Container-sharing techniques have been vastly proposed in recent years to better serve serverless computing [23, 36, 37, 40, 50]. Despite existing works claiming to share containers among functions securely, one may question *RainbowCake* for compromising security by sharing Lang and Bare containers. In *RainbowCake*'s design, we strictly follow serverless computing workflow and reverse the steps of container startup to enable safe container sharing. All the functions are enforced to run on *RainbowCake* as non-root users of containers [10, 11, 28]. For any functions, we first create and anonymously amount an `Action` folder into the container before importing user packages. We install user dependencies and manage all user files under the `Action` folder. Before downgrading to `Lang`, we enforce the container to remove the `Action` folder by sending a `Clean` HTTP request. We prevent attackers from breaching their workspace with jail techniques [35], such as `chroot` jail in Linux systems [16]. User-specified libraries are treated as part of the `User` layer, and user-spawned background processes can be tracked by recording the PIDs upon execution. We make sure `Lang` and `Bare` containers are isolated from any user-related contents by removing them before sharing the container with other functions. *RainbowCake* guarantees that a `User` container's user remnants are completely wiped out before moving to `Lang` and `Bare`. Hence, we assure that all `Lang` and `Bare` containers of *RainbowCake* are safe for cross-function sharing.

## 6 Implementation

*RainbowCake* is designed to be a generic solution to mitigate cold-starts and reduce memory wasting for serverless platforms. For concreteness, we describe its implementation in the context of Apache OpenWhisk [6], an open-source, distributed serverless platform that executes functions using Docker containers [41]. We implement *RainbowCake* with 8K lines of Scala for OpenWhisk modifications and 2K lines of Python for user client that runs experiments, which is open-sourced[3] for community adoption.

### 6.1 OpenWhisk's Container System

We implement our layer-wise policy on top of the OpenWhisk container system. OpenWhisk employs the actor model [32] to construct the container system using Akka Actor library [38]. On each worker server, OpenWhisk first initializes a parent actor to represent the *container pool*, which further spawns on-demand child actors to represent *container proxy* driven by function invocations. Upon receiving a function invocation, OpenWhisk forwards the invocation message to the container pool actor. The container pool checks whether it can reuse a warm container. If no warm container is found, the container pool creates a new container proxy actor, which initializes a new Docker container with a specific language runtime and supervises it to execute user functions.

OpenWhisk manages life cycles of containers by wrapping every container proxy into a Finite State Machine (FSM). Fig. 5 illustrates the diagram of the states of an OpenWhisk container proxy with *RainbowCake*'s layer-wise policy. In *RainbowCake*, a container upgrades or downgrades to the next type when its proxy transitions into the next state. A container proxy keeps its supervised container active and listens to HTTP requests for initializing function code, running the function, returning execution results to container FSMs, and terminating the container. In OpenWhisk, the cold startup overhead of a function invocation includes preparing an Akka actor, initializing the language runtime, and loading user package in the container. *RainbowCake* further divides three container types based on the initialization stages observed in OpenWhisk.

### 6.2 Layer-wise Policy Implementation

We focus on designing server-level pre-warming and keep-alive policy, *i.e.*, we do not discuss scheduling or load balancing among multiple servers since they are orthogonal. In our *RainbowCake*'s system architecture, we have one container pool that manages life cycles of multiple containers. To implement our layer-wise policy in OpenWhisk, we modified the following modules in OpenWhisk:

**HTTP handlers**. OpenWhisk implements a daemon that routes HTTP requests inside all its language runtimes, where the request type is specified in the URL path. Upon receiving an `Init` HTTP request, the container proxy first creates an `Action` folder under user home directory and installs user packages (import user code from database, activate libraries, and set environment variables) under it. We add a `Clean` HTTP handler to delete the `Action` directory and unset all user-defined environment variables. When a `User` container downgrades to `Lang`, it makes an HTTP POST request specifying `Clean` in the URL path to wipe out user packages, thus returning to a plain language runtime container. All three HTTP handlers are implemented in OpenWhisk language runtimes including Java, Python, and Node.js.

**Container proxy**. We make container proxies communicate with the container pool to query new TTLs before entering the next keep-alive period. The proxy operates its container using the HTTP handlers and Docker APIs [41]. Because of using actor models, the inter-transition overhead is negligible compared to the installing time of three layers (§7.7).

**Container pool**. The invocation history recorder and distribution modeling are implemented in the OpenWhisk container pool. We use Linux `cgroups` tools [31] to measure the memory occupation of containers. The maintenance overhead of the history recorder is trivial, as the total memory footprint of keeping track of one million functions only requires 250 MB of memory. For pre-warming a container, the container pool queries the history recorder for the given function and estimates the IAT of the next invocation arrival. We then register a pre-warming event using the utility method `scheduleOnce(IAT, pre-warm)` from Akka Schedulers, which pre-warm a `User` container after waiting for an IAT duration. For determining keep-alive TTLs, when a container proxy needs to enter the next keep-alive period, it first sends a query embedded with its current container properties (*e.g.*, layer name, user package ID, and language name)

[3] https://github.com/IntelliSys-Lab/RainbowCake-ASPLOS24

to the container pool. The pool estimates an IAT from the history recorder with a corresponding invocation distribution and sends the IAT back to proxies as the new TTL.

## 7 Evaluation

### 7.1 Experimental Setup

**Testbeds.** We deploy and evaluate *RainbowCake* on a three-node OpenWhisk cluster using AWS EC2 [7], including one client node for invoking functions, one controller node that hosts OpenWhisk components, and one worker node. The `m5.4xlarge` client node has 16 AMD EPYC CPU cores, the `m5.8xlarge` controller node has 32 CPU cores, and the worker of `m5.24xlarge` has 96 CPU cores and 240 GB of memory for launching Docker containers and executing functions. Each of the three nodes has 100 GB EBS storage.

**Workloads.** We employ three open-source serverless benchmark suites [17, 18, 52] with in total 20 applications varying from Node.js, Python, and Java in our evaluation. Table 1 characterizes the 20 functions across three programming languages and five domains. We configure each function's resource limit and input data according to the default settings from the suites.

**Invocation traces.** We used real-world Azure Functions invocation traces [53] for evaluation on OpenWhisk. In total, eight trace sets were sampled for evaluating *RainbowCake*. Specifically, we sampled an 8-hour trace set for evaluating the overall performance, and other seven 1-hour trace sets with IAT CVs ranging from 0.2 to 4.0, to evaluate *Rainbow-Cake* 's robustness to workloads with different IATs. For sampling each set, we swipe the 14-day Azure invocation trace files and select the function traces that first match our required invocation IAT CVs. Each trace is mapped to one function and drives the invocations in evaluation. We leave the sampled traces unaltered to ensure an evaluation closely reflecting production environments.

**Table 1.** Characterizations of serverless applications.

| Language | Function | Domain |
|---|---|---|
| Node.js | Auto Complete (AC) | Web App |
| | Dynamic HTML (DH) | Web App |
| | Uploader (UL) | Web App |
| | Image Sizing (IS) | Multimedia |
| | Thumbnailer (TN) | Multimedia |
| | OCR-Image (OI) | Multimedia |
| Python | DNA Visualization (DV) | Scientific Computing |
| | Graph BFS (GB) | Scientific Computing |
| | Graph MST (GM) | Scientific Computing |
| | Graph Pagerank (GP) | Scientific Computing |
| | Image Recognition (IR) | Machine Learning |
| | Sentiment Analysis (SA) | Machine Learning |
| | File Compression (FC) | Web App |
| | Markdown (MD) | Web App |
| | Video Processing (VP) | Multimedia |
| Java | Data Transform (DT) | Data Analysis |
| | Data Load (DL) | Data Analysis |
| | Data Query (DQ) | Data Analysis |
| | Data Scan (DS) | Data Analysis |
| | Data Group (DG) | Data Analysis |

***RainbowCake*'s settings.** *RainbowCake* can configure three important system parameters for improving performance: knob parameter $\alpha$ that balances the initialization cost and memory waste cost, window size $n$ that keeps historical information of recent invocations, and the quantile $p$ that represents the confidence for estimating invocation IATs. In our evaluation, we compute the upper bounds, which limit the maximum survival time of idle containers, for the three container types per function and set them as the initial TTLs of the three container types. When the experiment proceeds, the TTLs will be dynamically configured in runtime using Eq. 7. We set the knob parameter $\alpha$ of unified cost as 0.996 so that all functions' initialization cost consistently outweighs the memory waste cost. The invocation history recorder uses a sliding window to monitor the latest $n$ invocations for modeling distributions per container type. The window size $n$ is set to six invocations. We set all functions' IAT quantile to be 0.8. The sensitivity of *RainbowCake*'s three parameters are analyzed in §7.5.

**Baselines.** We compare *RainbowCake* with five state-of-the-art cold-start mitigation techniques: 1) **OpenWhisk**, the default keep-alive policy in OpenWhisk that keeps every idle container alive for a fixed 10 minutes before termination. Commercial serverless platforms such as AWS Lambda, Google Cloud Function, and Azure functions adopt a similar strategy. 2) **Histogram** [53], a histogram-based full container caching approach to dynamically adjust pre-warming and keep-alive TTLs by predicting the inter-arrival time of function invocations. 3) **FaaSCache** [25], a greedy full container caching approach to predict and decide which container to keep-alive, where containers are cached alive in the pool until evicted due to creating new containers. 4) **SEUSS** [13], a partial container caching approach to handle invocation bursts. Since the code repository of SEUSS [51] is outdated (not maintained since 2019), we implement SEUSS in OpenWhisk for evaluation. 5) **Pagurus** [37], a container sharing scheme to recycle idle containers for helping other functions. Pagurus claimed to use AWS best-practice applications, yet the implementations are all sleep functions [44]. Original Pagurus and its sleep functions cannot provide meaningful utilization metrics, so we implement Pagurus in OpenWhisk and evaluate it with realistic serverless functions.

### 7.2 Latency and Memory Waste

We evaluate the performance of OpenWhisk, Histogram, FaaSCache, SEUSS, Pagurus, and *RainbowCake* using the 8-hour Azure trace set and 20 serverless functions. Fig. 10 characterizes the total invocation arrivals of the 8-hour trace set in minutes. The Azure Functions dataset originally depicts invocations in per-minute buckets. When replaying the traces, we inject the invocation at the beginning of the

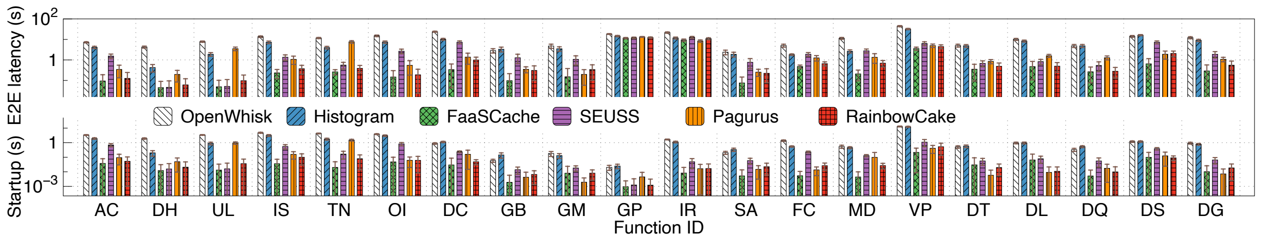

**Figure 6.** Average function startup and end-to-end latency of six baselines.

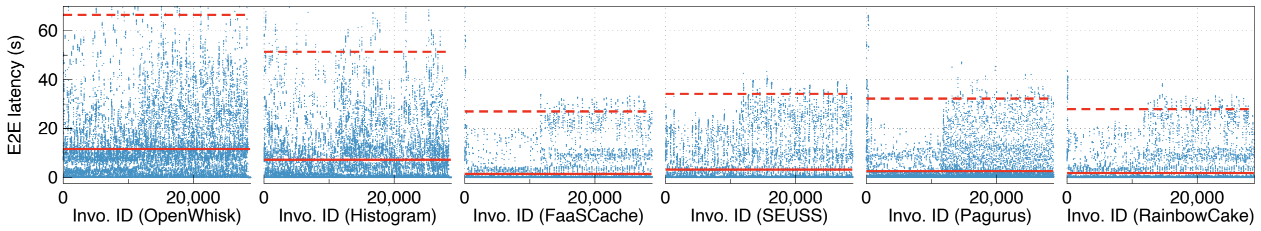

**Figure 7.** End-to-end latency of each invocation executed by six baselines. Red dash and solid lines represent the 99th percentile and average latency, respectively.

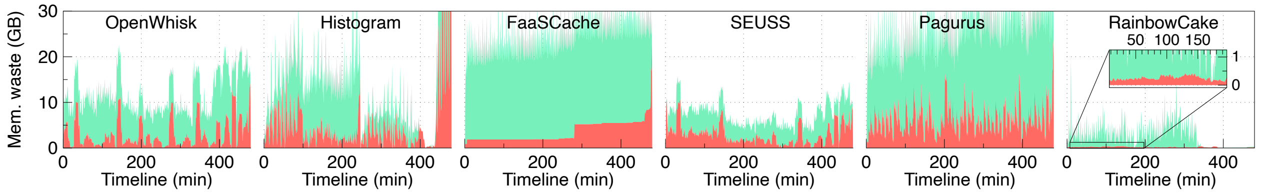

**Figure 8.** Timeline of wasted memory of six baselines. Green and red shadows represent memory wasted but eventually hit and memory wasted never hit by invocations, respectively.

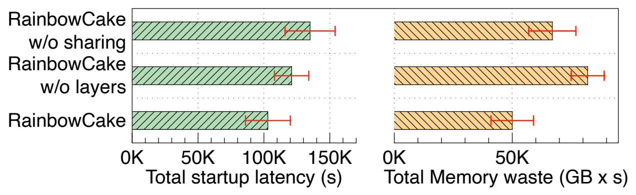

**Figure 9.** Ablation study of *RainbowCake*.

minute if only one invocation exists in a minute-bucket; otherwise, for multiple invocations within a minute, we evenly distribute invocations throughout the minute, similar to the methodology used in FaaSCache [25].

**Function latency**. Fig. 6 shows the average end-to-end (top) and startup (bottom) latency per function for six baselines, respectively. Compared to OpenWhisk, Histogram, SEUSS, and Pagurus, *RainbowCake* reduces the average end-to-end/startup latency by 69%/97%, 60%/96%, 43%/74%, and 31%/68%, respectively. *RainbowCake* increases the average function end-to-end/startup latency by 50ms/20ms compared to FaaSCache.

**Invocation details**. Fig. 7 shows the end-to-end latency of every invocation executed by six baselines. Red dash and straight lines represent the 99th percentile (P99) and average latency. Compared to OpenWhisk, Histogram, SEUSS, and Pagurus, *RainbowCake* reduces the average/P99 invocation end-to-end latency by 84%/58%, 75%/45%, 43%/18%, and 29%/13%, respectively. *RainbowCake* increases the average/P99 invocation end-to-end latency by 0.4s/1.8s compared to FaaSCache.

**Memory waste**. Fig. 8 shows the timeline of memory waste for six baselines. Compared to OpenWhisk, Histogram, FaaSCache, SEUSS, and Pagurus, *RainbowCake* reduces total memory waste by 60%, 63%, 75%, 44%, and 77%, respectively. FaaSCache incurs excessive memory waste throughout the experiment due to no container termination. Pagurus also has considerable memory waste because of heavy monolithic containers.

## 7.3 Ablation Study

We conduct an ablation study by comparing *RainbowCake* with its two variants: 1) ***RainbowCake* without sharing-aware modeling.** We replace the sharing-aware modeling with a fixed keep-alive TTL policy similar to the OpenWhisk default policy. We set the 5, 3, and 2 minutes for `User`, `Lang`, and `Bare` keep-alive TTLs, respectively. 2) ***RainbowCake* without layer caching.** We only pre-warm and keep `User` containers alive, and terminate them once timed out to skip `Bare` and `Lang` phases. We run the experiments using the same traces and functions in §7.2.

**Startup latency.** Left of Fig. 9 shows the total startup latency summed over all invocations. *RainbowCake* increases

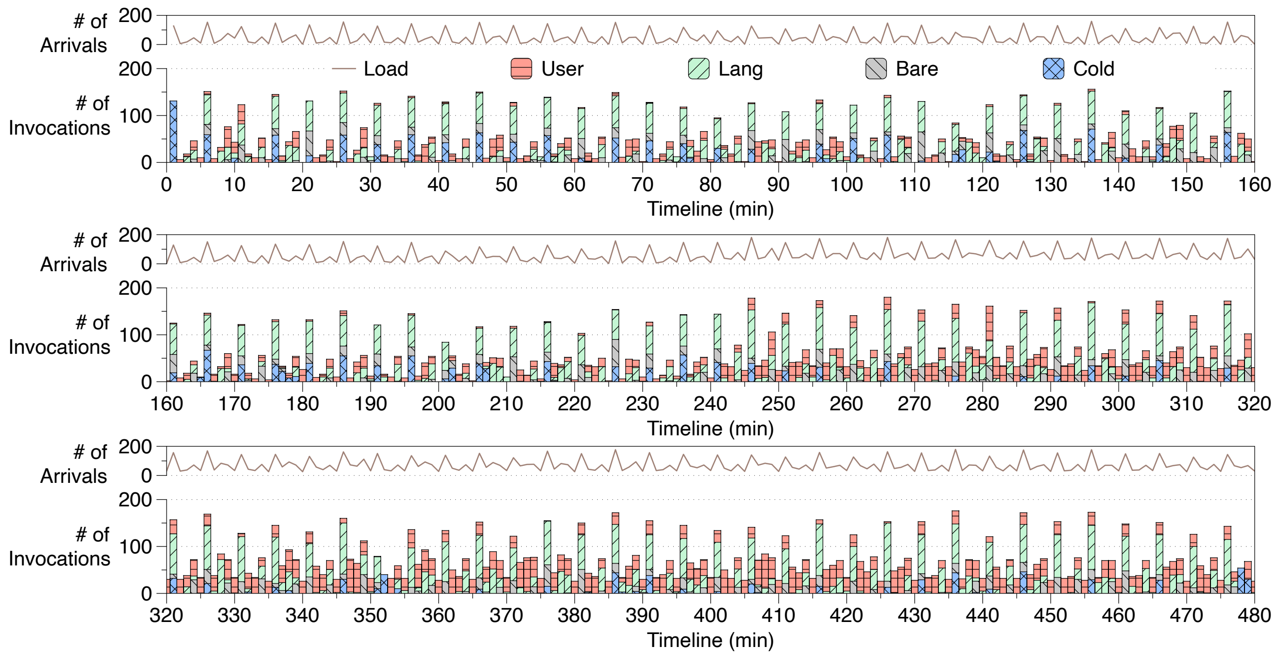

**Figure 10.** Invocation arrivals and startup timeline of the 8-hour trace set with *RainbowCake*.

the total startup latency by 23% and 14% without sharing-aware modeling and layer caching, respectively.

**Memory waste.** Right of Fig. 9 shows the total memory waste time of invocations. *RainbowCake* increases the memory wasting by 25% and 39% without sharing-aware modeling and layer caching, respectively.

## 7.4 Performance Source Analysis

Fig. 10 characterizes the invocation arrivals and the number of invocations in each startup type of the 8-hour trace set with *RainbowCake* in minutes. The number of cold-starts is significantly reduced, as many cold invocations are offloaded to shareable `Lang` and `Bare` containers. While `User` containers reduce 35% cold-starts, `Lang` and `Bare` containers also reduce 41% and 13%, respectively. The analysis shows that reusing all three types of containers is necessary for *RainbowCake*'s cold-start mitigation.

## 7.5 Sensitivity Analysis

We analyze the sensitivity of three parameters in *RainbowCake*: knob parameter $\alpha$ of unified cost in Eq. 1, IAT quantile $p$ in Eq. 4, and the size of invocation sliding window $n$ in Eq. 5. We use the same traces and functions in §7.2 to run the experiments. Recall that in Eq. 1, *unified cost* consists of the total initialization cost (startup latency) and the total memory cost from (memory wasting). We show the contribution of each cost in Fig. 11.

**Knob parameter** $\alpha$. We set $\alpha = 0.996$ in Eq. 1 so that the total initialization cost outweighs the total container memory wasting cost. Fig. 11(a) shows the unified cost when gradually increasing $\alpha$ from 0.990 to 0.999 in the step of 0.001.

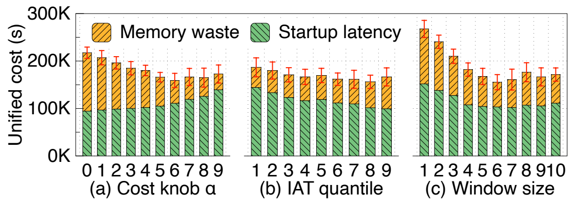

**Figure 11.** Sensitivity analysis of *RainbowCake*'s total wasting cost and total startup cost for knob parameter $\alpha$ (0.990 to 0.999), IAT quantile $p$ (0.1 to 0.9), and invocation sliding window $n$ (1 to 10).

With $\alpha$ increasing, the total wasting cost contributes less to the unified cost while the total initialization cost grows. *RainbowCake* achieves the lowest cost when $\alpha$ is 0.996.

**IAT quantile** $p$. In our evaluation, we set *RainbowCake*'s IAT quantile as 0.8 for determining keep-alive TTLs. Fig. 11(b) shows the unified cost when gradually increasing $p$ from 0.1 to 0.9 in the step of 0.1. When $p$ increases, *RainbowCake* gradually predicts the IATs wildly with longer keep-alive TTLs, which increases the total wasting cost. Meanwhile, longer keep-alive TTLs mitigate cold-starts and decreases the total initialization cost. *RainbowCake* achieves the lowest unified cost when $p$ is set to 0.8.

**Size of invocation sliding window** $n$. We set the size of the sliding window as six invocations. Fig. 11(c) reports the unified cost when increasing the window size from 1 to 10 in the step of 1. When the size increases from 1 to 5, the characteristics that *RainbowCake* captures from functions become more precious. After six invocations, the latest invocation patterns are interfered with by staled invocation information. *RainbowCake* achieves the lowest unified cost when $n$ is set to six invocations.

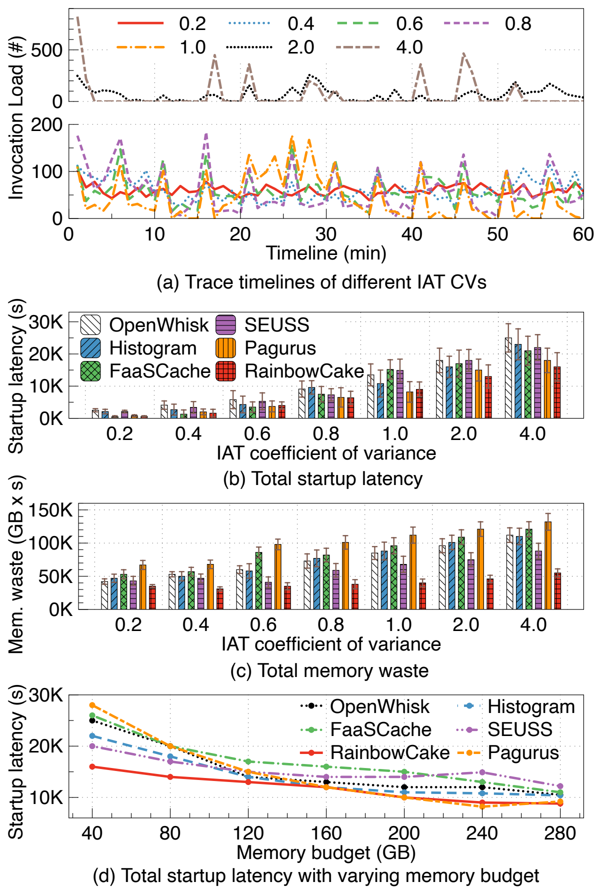

**Figure 12.** Robustness to burstiness and limited memory budgets of six baselines.

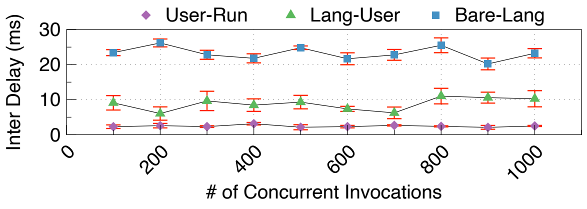

**Figure 13.** Inter-transition overhead as the number of concurrent invocations increase, including `Bare-to-Lang` (B-L), `Lang-to-User` (L-U), and `User-to-Run` (U-Run).

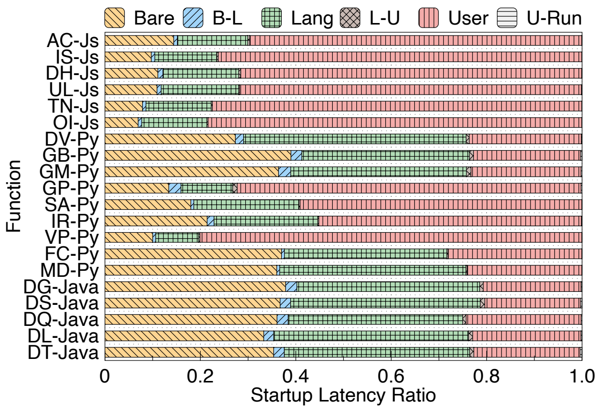

**Figure 14.** Relative ratio of startup latency breakdown of 20 functions used in the evaluation, including three layers and inter-transition overheads: `Bare-to-Lang` (B-L), `Lang-to-User` (L-U), and `User-to-Run` (U-Run).

## 7.6 Robustness to Burstiness and Concurrency

We examine the robustness of *RainbowCake* and five baselines when dealing with burstiness and concurrency. We sampled seven sets of 1-hour invocation traces from the Azure Functions trace dataset, where the trace sets have a Coefficient of Variance (CV) of IAT from 0.2 to 4.0. Each trace set contains 3,600 function invocations. Fig. 12(a) characterizes the invocation arrival timelines of seven trace sets. With a higher IAT CV, a trace is more bursty and volatile [53]. Intuitively, a trace with an IAT CV of 0 invokes the same number of invocations every minute, whereas a trace with an IAT CV of 4 is full of burstiness. *RainbowCake* is more robust to burstiness than five baselines due to joint support of partial container caching and sharing. We analyze the details of the results below.

**Startup latency.** Fig. 12(b) shows the total startup latency of invocations handled by *RainbowCake* and five baselines. *RainbowCake* is more robust to burstiness while maintaining the slowest growth rate when the IAT CV increases.

**Memory wasting.** Fig. 12(c) shows the total memory waste time of invocations handled by *RainbowCake* and five baselines. We use the product of memory occupation and idle time to characterize overall memory wasting. *RainbowCake* achieves significantly less memory wasting than five baselines when the IAT CV increases.

**Constrained memory budget.** Fig. 12(c) shows the total startup latency of invocations handled by *RainbowCake* and five baselines. We vary the memory size of the container pool to evaluate the robustness to constrained memory budget. *RainbowCake* shows significantly less total startup latency when the memory budget is limited.

**Concurrency**. Fig. 13 shows *RainbowCake*'s inter-transition overheads of `Bare` to `Lang` (B-L), `Lang` to `User` (L-U), and `User` to function execution (U-Run), respectively. We gradually increase the number of concurrent invocations from 100 to 1,000. All three inter-transition overheads consistently stay trivial compared to function startup latency and execution time, with negligible fluctuations when the number of concurrent invocations increases.

## 7.7 Overheads of *RainbowCake*

Fig. 14 shows the startup latency breakdown of 20 real-world functions used in the evaluation. Each part is normalized to a relative percentage compared to the total startup latency. Latency components include initialization delay for three layers (`Bare`, `Lang`, and `User`) and the inter-transition overheads (B-L, L-U, U-Run), respectively. The total inter-transition overhead (B-L + L-U + U-Run) incurs less than 3% delay, which is negligible for the total startup latency, while providing extensive merits such as cold starts mitigation via container sharing and memory wasting reduction.

### 7.8 Integrating with Orthogonal Techniques

*RainbowCake* can be easily integrated with orthogonal techniques, such as checkpointing [21, 55], to accelerate container cold startup. We evaluate checkpoint-support *RainbowCake* by enabling Docker container checkpoint API [20] in OpenWhisk, which relies on CRIU [19] to checkpoint and restore container processes. Instead of launching new containers from scratch, checkpoint-support *RainbowCake* can restore a container from checkpoint files. When running the same experiment as in §7.2, checkpoint-support *RainbowCake* reduces the average startup latency by 36% while increasing the total memory waste by 15% due to caching additional checkpoint images in memory.

## 8 Discussion

***RainbowCake* on distributed clusters.** To enable *RainbowCake* across multi-node distributed clusters, we can design an inter-node scheduler considering the three factors: 1) *Locality*—prioritizing a node with fully warmed (`User`) containers, 2) *Sharing*—selecting a node with the most available layer-sharing (`Lang` and `Bare`) opportunity, and 3) Load—distributing functions in a manner that avoids resource contention and waste.

***RainbowCake* with tiered caching.** To integrate *RainbowCake* with tiered caching, we can leverage the characteristics of different caches. For example, with memory and non-volatile memory (NVM), we can adaptively cache different layers in memory and NVM by computing their priorities using statistics, such as hit rate and memory footprint. Frequently-hit or heavy layers can be cached in memory for fast access while storing others in NVM.

**Security of *RainbowCake*'s container sharing.** *RainbowCake*'s container sharing may introduce two types of user data leakage: 1) The attacker creates a malicious container by invoking its malicious function, where *RainbowCake* may fail to clean the function remnants such as background processes. 2) The attacker invokes its function to obtain a container, purposely detects if a container is shared or reused, and searches for any states (such as code, input data, dependencies) left by previous users. In *RainbowCake*, we can snapshot `Bare` and `Lang` containers as zygotes and serve functions by forking the zygote templates [21, 37]. The zygote templates are safe checkpoints, as they do not import any user-related code and data. Hence, the privacy of function software environments can be enhanced.

## 9 Related Work

**Virtualization-layer solutions.** Recently, researchers have focused on virtualization-layer infrastructure optimization to provide faster container initialization. Some works develop new container types [1, 27] and propose to initialize containers from checkpoints [21, 55]. Other works develop efficient memory management techniques to accelerate container startup [5, 49]. *RainbowCake* is orthogonal to virtualization-layer solutions and can be easily integrated with them.

**Container-caching solutions.** Apart from low-level improvements, massive works aim to mitigate function startup latency via full container caching [12, 15, 25, 29, 45, 47, 48, 53]. Recent research proposes partial container caching [13, 39, 42] to alleviate coarse-grained trade-offs in full container caching. However, experimental results show that *RainbowCake* outperforms existing partial container caching solutions by reducing function startup latency with sharing awareness.

**Container-sharing solutions.** Under the premise of not compromising security, container-sharing techniques have become popular for optimizing serverless computing and alleviating cold-starts [2, 23, 37, 40, 43, 50]. *RainbowCake* outperforms state-of-the-art container sharing techniques by achieving similar (or better) function startup latency while minimizing memory waste. Commercial serverless platforms' shareable layers, such as AWS Lambda layers [8], are designed for fast user code deployment, not for mitigating cold-starts.

## 10 Conclusion

This paper proposed *RainbowCake*, a layer-wise container pre-warming and keep-alive technique with sharing awareness that effectively mitigates cold-starts with minimal memory waste. Compared to existing solutions, *RainbowCake* is tested to be more robust and tolerant to function invocation bursts on real clusters with realistic workloads. Experimental results showed that *RainbowCake* reduces 68% startup latency and 77% container memory waste compared to existing solutions.

## 11 Acknowledgements

We thank our shepherd Dr. Rodrigo Bruno and anonymous reviewers for their valuable input. The work of H. Yu, C. Fontenot, and H. Wang was supported in part by NSF CRII-OAC-2153502 and the AWS Cloud Credit for Research program. The work of R. Basu Roy and D. Tiwari was supported in part by NSF CNS-CORE-Small-1910601 and CNS-CORE-Small-2124897. The work of J. Li was supported in part by NSF CRII-CNS-NeTS-2104880 and RINGS-2148309, as well as ARO W911NF-23-1-0072, and DOE DE-EE0009341. The work of S. Park was supported by NSF 2120248. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

## References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proc. the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[2] 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 *Proc. the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, 2018.

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. BATCH: Machine Learning Inference Serving on Serverless Platforms With Adaptive Batching. In *Proc. of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2020.

[4] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2018.

[5] Lixiang Ao, George Porter, and Geoffrey M Voelker. FaaSnap: FaaS Made Fast Using Snapshot-Based VMs. In *Proc. the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[6] Apache. Apache OpenWhisk Official Website. https://openwhisk.apache.org, 2018.

[7] AWS. AWS EC2: Secure and Resizable Compute Capacity in the Cloud. https://aws.amazon.com/ec2/, 2020. [Online; accessed 1-May-2020].

[8] AWS. AWS Lambda Layers. https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html, 2022. [Online].

[9] AWS. AWS Lambda Runtimes. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html, 2022. [Online].

[10] AWS Lambda. Troubleshooting AWS Lambda Identity and Access. https://docs.aws.amazon.com/lambda/latest/dg/security_iam_troubleshoot.html, 2022. [Online; accessed 1-April-2022].

[11] Azure Functions. Authentication and Authorization in Azure Functions. https://learn.microsoft.com/en-us/azure/app-service/overview-authentication-authorization, 2022. [Online; accessed 1-April-2022].

[12] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *Proc. the ACM Symposium on Cloud Computing (SoCC)*, 2021.

[13] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proc. the European Conference on Computer Systems (EuroSys)*, 2020.

[14] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2019.

[15] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2021.

[16] Bill Cheswick. An Evening with Berferd in which a cracker is Lured, Endured, and Studied. In *Proc. Winter USENIX Conference, San Francisco*, 1992.

[17] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference (Middleware)*, 2021.

[18] Robert Cordingly, Hanfei Yu, Varik Hoang, David Perez, David Foster, Zohreh Sadeghi, Rashad Hatchett, and Wes J Lloyd. Implications of Programming Language Selection for Serverless Data Processing Pipelines. In *Proc. of the 2020 IEEE Conference on Cloud and Big Data Computing (CBDCom)*, 2020.

[19] CRIU. CRIU-Docker. https://criu.org/Docker. [Online].

[20] Docker. Docker Checkpoint API. https://docs.docker.com/engine/reference/commandline/checkpoint/. [Online].

[21] 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 *Proc. the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[22] Alexander N Dudin, Valentina I Klimenok, and Vladimir M Vishnevsky. *The Theory of Queuing Systems with Correlated Flows*. Springer, 2020.

[23] Tarek Elgamal. Costless: Optimizing Cost of Serverless Computing Through Function Fusion and Placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018.

[24] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proc. the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[25] Alexander Fuerst and Prateek Sharma. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proc. the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[26] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. Reducing Latency via Redundant Requests: Exact Analysis. *ACM SIGMETRICS Performance Evaluation Review*, 2015.

[27] Google. gVisor. https://gvisor.dev/, 2018. [Online; accessed 2-May-2018].

[28] Google Cloud Functions. Function Identity in Google Cloud Functions. https://cloud.google.com/functions/docs/securing/function-identity, 2022. [Online; accessed 1-April-2022].

[29] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T Kandemir, and Chita R Das. Fifer: Tackling Underutilization in the Serverless Era. In *Proc. the 21st International Middleware Conference (Middleware)*, 2020.

[30] Wang Hao, Niu Di, and Li Baochun. Distributed Machine Learning with a Serverless Architecture. In *Proc. the IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[31] Heo, Tejun. Control Group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt, 2021. [Online; accessed 1-April-2022].

[32] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. the International Joint Conference on Artificial Intelligence (IJCAI)*, 1973.

[33] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proc. 2017 Symposium on Cloud Computing (SoCC)*, 2017.

[34] Gauri Joshi, Yanpei Liu, and Emina Soljanin. On the Delay-storage Trade-off in Content Download from Coded Distributed Storage Systems. *IEEE Journal on Selected Areas in Communications (JSAC)*, 2014.

[35] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the Omnipotent Root. In *Proceedings of the 2nd International SANE Conference*, 2000.

[36] Seungjun Lee, Daegun Yoon, Sangho Yeo, and Sangyoon Oh. Mitigating Cold Start Problem in Serverless Computing with Function Fusion. *Sensors*, 2021.

[37] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through {Inter-Function} Container Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC)*, 2022.

[38] Lightbend. Introduction to Actors: Akka Documentation. https://doc.akka.io/docs/akka/current/typed/actors.html, 2021. [Online; accessed 1-Aug-2021].

[39] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes. In *Proc. the 11th Workshop on Programming Languages and Operating Systems (PLOS)*, 2021.

[40] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. {ORION} and the Three Rights: Sizing, Bundling, and Prewarming for Serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[41] Dirk Merkel et al. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014.

[42] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[43] 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 *Proc. the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, 2018.

[44] Pagurus. Pagurus Code. https://github.com/lzjzx1122/Pagurus, 2023.

[45] Li Pan, Lin Wang, Shutong Chen, and Fangming Liu. Retention-Aware Container Caching for Serverless Edge Computing. *Proc. of IEEE Conference on Computer Communications (INFOCOM)*, 2022.

[46] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Mashup: Making Serverless Computing Useful for HPC Workflows via Hybrid Execution. In *Proc. of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2022.

[47] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. DayDream: Executing Dynamic Scientific Workflows on Serverless Platforms with Hot Starts. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[48] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *Proc. the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[49] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory Deduplication for Serverless Computing with Medes. In *Proc. the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[50] Trever Schirmer, Joel Scheuner, Tobias Pfandzelter, and David Bermbach. FUSIONIZE: Improving Serverless Application Performance Through Feedback-driven Function Fusion. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, 2022.

[51] SEUSS. SEUSS Code. https://github.com/SESA/SEUSS, 2023.

[52] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural Implications of Function-as-a-Service Computing. In *Proc. the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[53] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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 *Proc. the USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[54] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless Linear Algebra. In *Proc. of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020.

[55] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking Functions to Warm the Serverless Cold Start. In *Proc. the 21st International Middleware Conference (Middleware)*, 2020.

[56] Yin Sun, Zizhan Zheng, C Emre Koksal, Kyu-Han Kim, and Ness B Shroff. Provably Delay Efficient Data Retrieving in Storage Clouds. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[57] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the Curtains of Serverless Platforms. In *Proc. the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, 2018.