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

Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, and Chuhao Xu, *Shanghai Jiao Tong University;* Deze Zeng, *School of Computer Science, China University of Geosciences;* Zhuo Song, Tao Ma, and Yong Yang, *Alibaba Cloud;* Chao Li and Minyi Guo, *Department of Computer Science and Engineering, Shanghai Jiao Tong University*

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

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

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

[1,3]Zijun Li, [1]Linsong Guo, [1]Quan Chen, [1]Jiagan Cheng, [1]Chuhao Xu, [2]Deze Zeng, [3]Zhuo Song, [3]Tao Ma, [3]Yong Yang, [1]Chao Li, [1]Minyi Guo
[1]*Department of Computer Science and Engineering, Shanghai Jiao Tong University*
[2]*School of Computer Science, China University of Geosciences*
[3]*Alibaba Cloud*

Abstract

In serverless computing, each function invocation is executed in a container (or a Virtual Machine), and container cold startup results in long response latency. We observe that some functions suffer from cold container startup, while the warm containers of other functions are idle. Based on the observation, other than booting a new container for a function from scratch, we propose to alleviate the cold startup by re-purposing a warm but idle container from another function. We implement a container management scheme, named **Pagurus**, to achieve the purpose. Pagurus comprises an intra-function manager for replacing an idle warm container to be a container that other functions can use without introducing additional security issues, an inter-function scheduler for scheduling containers between functions, and a sharing-aware function balancer at the cluster-level for balancing the workload across different nodes. Experiments using Azure serverless traces show that Pagurus alleviates 84.6% of the cold startup, and the cold startup latency is reduced from hundreds of milliseconds to 16 milliseconds if alleviated.

1 Introduction

Owing advantages of high maintainability and testability, serverless computing is suitable for the ever-growing Internet services (the tenants are charged per invocation). As a result, hyperscalers now provide serverless computing services (e.g., Amazon Lambda [5], Google Cloud Function [11], Microsoft Azure Functions [12], and Alibaba Function Compute [1]). Meanwhile, some open-source serverless computing solutions like Apache OpenWhisk [3] and Fission [9] have also been developed and released.

In serverless computing, user functions run in containers (or Virtual Machines), and the containers are specialized for a function (a container is not allowed to run different user functions). Warm containers refer to the keep-alive containers that serve subsequent invocations (*warm startup*). If there is no warm container for a function invocation, a new container is started from scratch (*cold startup*). The cold startup

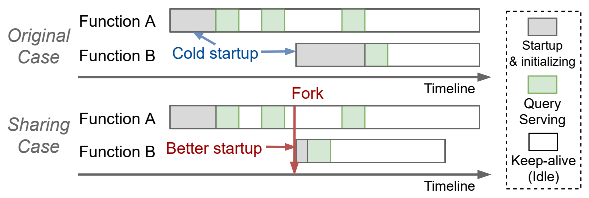
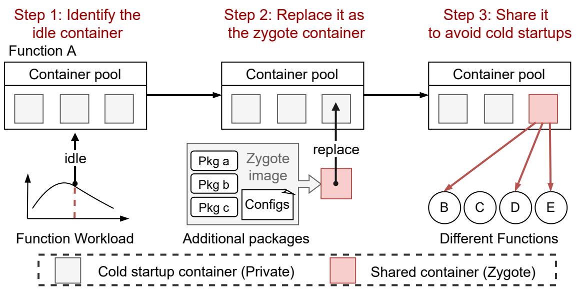


Figure 1: “Fork” an idle container of Function *A* for Function *B* to alleviate its cold startups.

latency is more than $10\times$ of the warm startup latency due to the container creation, software environment setup, and code initialization [40,44,47,49,59,61]. It is ideal if all functions can run in warm containers. However, keep-alive containers are recycled to save resources once no new invocation arrives during the lifetime.

Many efforts have been devoted to speeding up the container cold startup [7, 28, 31, 32, 45, 46, 54–56, 58]. The prewarm-based methods create containers and runtime in advance, one method of which is prewarming customized containers for each function that includes all its required software packages [8, 15, 27, 28, 60]. However, it brings heavy memory consumption. Another method of prewarming containers is only installing common packages, and all functions share the prewarmed container pool [14,45,46]. This method is more memory space-efficient, but generating customized containers for a function from the prewarmed containers suffers from package download and installing latency overhead. Current solutions mainly adopt the second method [14,46].

To alleviate the memory waste and minimize the function response time, instead of prewarming containers, we propose to alleviate the cold container startup through container sharing. For instance, if function *A* in Figure 1 can “fork or lend” the runtime checkpoint from its idle warm containers for function *B*, the cold startup of *B* is eliminated. By such means, we can leverage a function’s idle warm containers before being recycled to help others that tend to experience cold container startup.



Step 1: Identify the idle container
Step 2: Replace it as the zygote container
Step 3: Share it to avoid cold startups
Function A
Container pool
Container pool
Container pool
idle
replace
Pkg a
Pkg b
Pkg c
Zygote image
Configs
B
C
D
E
Function Workload
Additional packages
Different Functions
Cold startup container (Private)
Shared container (Zygote)

Figure 2: The container sharing logic and challenging steps.

Analyzing the container sharing procedure, we find three challenges in achieving such goal. **1) The function invocation loads are not stable [51].** It is difficult to identify whether function *A*'s warm containers are actually idle or not. If the warm containers of *A* are always used to help *B*, *A*'s invocation may not be able to get a warm container and may even suffer from Quality-of-Service (QoS) violation. **2) Functions rely on different software packages.** When function *A* "forks" a warm container for function *B*, the warm container has to install extra packages. Installing and importing these packages may take longer than the cold container startup. Worse still, greedy re-packaging for excessive functions can also lead to huge image size or incur package version contradictions. **3) "Forking" an idle container from other functions may introduce security vulnerabilities.** While a function's code or data are stored privately, sharing the container with other functions is risky.

We propose a container management system, named **Pagurus** to tackle these challenges. Figure 2 shows the steps of using function *A*'s idle warm containers to alleviate the cold container startup of other functions. The key idea is to replace its idle containers with new containers that other functions can safely use. The new container is created through a new image that already installs the required software packages of other functions, without including the code/data of the original function *A*. In this way, the warm containers of a function are classified into three categories: *private containers*, *zygote containers*, and *helper containers*. A function's private containers can only be used by itself. Its zygote containers are the new containers that other functions can use. Its helper containers are the containers forked from other functions' zygotes. A helper container already loads its user code for future invocations. By using privilege control in the operating system, a function that uses a helper container cannot obtain any code, data, or package information of other functions.

Pagurus uses an *intra-function container manager* for each function to manage its containers, an *inter-function scheduler* on each node to manage the "fork" action between functions, and a *sharing-aware function balancer* to schedule functions across the nodes. For a function, the intra-function manager monitors the status of each container, identifies idle warm containers, and re-purposes an idle container based on a QoS-based timer. The inter-function scheduler, acting as an orchestrator, determines the to-be-helped functions of each function. We design a Similarity Filtered Weighted Random Sampling (SF-WRS) algorithm to find an appropriate set of to-be-helped functions. Besides, the sharing-aware function balancer distributes function invocations to different nodes to achieve efficient inter-function container sharing.

Pagurus requires no offline analysis or profile on the functions, thus can be easily adopted in production. The main contributions of this paper are as follows.

- **A resource-friendly design of zygote and helper container.** Zygote container enables resource-saving through package and function reclamation, without incurring any additional security issues meanwhile.
- **The design of a SF-WRS re-packing policy.** Based on the package similarity between functions and the frequency of function cold startups, SF-WRS policy reduces the number of packages to-be-installed, thus minimizing the memory needed and the overhead of creating zygote containers.
- **The design of an efficient container sharing mechanism.** Pagurus divides the warm containers of a function into three types and manages the three types of containers in different ways. The mechanism efficiently alleviates the cold container startup.

We evaluate Pagurus using both best-practice AWS serverless functions [6] and Azure traces [50]. Experiments show that Pagurus alleviates 84.6% of cold startups on average in Azure traces, and the cold startup latency is reduced from hundreds of milliseconds to 16 milliseconds if alleviated.

2 Background and Related Work

If a function is invoked for the first time or there is no alive (or warm) container for it, the serverless system starts a new container to encapsulate its function runtime, initializes the software environment, loads application-specific code, and runs the function. All these steps make up a *cold startup* and may even take several seconds [21, 36, 59]. The cold startup significantly increases queries' end-to-end latency [23, 33, 47, 49]. The long latency problem worsens when the function invocation is short (e.g., hundreds of milliseconds).

Prewarm startup spawns template containers that are already initialized with the software environment. Though it skips the container startup and users only need to perform application-specific code initialization [2, 3, 32, 46], its pre-loaded packages can either make the image size too large [20, 32, 53], or cause more memory consumption for the prewarm container [24, 46, 50].

Many prior studies have been conducted to reduce the container startup latency [19, 28, 34, 48, 52, 54]. However, existing works mainly focused on seeking lightweight virtualization technologies to pursue lower overhead [17] or optimizing prewarm strategies for more accurate prediction models and less initialization cost []. A common optimization is to pause the container when idle to save resources consumed by function codes and packages, and then reload it for reusing when invoked [34, 44, 45, 57].

SAND [19] separated applications via containers while allowing functions of one application to run in the same container by different processes. FaasCache [31] took the caching model for objects into serverless context, and implemented the Greedy-Dual keep-alive caching mechanism to reduce the resource requirement and keep containers warm. Shahrad *et al.* [50] proposed to dynamically change the instance lifetime of the recycling and provisioning instances according to the time series prediction. Some researchers use C/R (Checkpoint and Restore) [7, 55, 56, 58] that restores container images from checkpoints to speed up the cold startup. For example, Catalyzer [28] utilized C/R to realize on-demand recovery. However, it still incurs long latency compared with a warm startup. The above technologies are orthogonal to us, and Pagurus can be combined with them to reduce the cold startup latency further. SOCK [46] introduced a tree cache and uses the benefit-to-cost model to update packages in the prewarmed containers dynamically, but the zygote design consumes more memory when maintaining packages by a cache-tree. Moreover, the cache-tree does not work if functions require conflicted package versions.

Pagurus resolves the problems through inter-function container sharing with conflict concerns, and needs neither pool-size tradeoffs nor time-consuming model training.

3 Investigation and Motivations

In this section, we discuss the current prewarm-based mechanism for alleviating the cold container startups, and show the possibility of eliminating the cold container startup with inter-function container sharing.

3.1 Latencies of Cold and Prewarm Startups

A cold container startup is done in three time-consuming steps: *create container from the image*, *initialize software environment*, and *initialize application-specific code*. With the prewarm mechanism (used in OpenWhisk [3] and production platforms), several containers that already import common libs/packages are hatched in a container pool. A function invocation with no warm container can specialize the prewarmed container by installing the extra packages.

We use prewarm-enabled OpenWhisk with local cache as the serverless platform, and use the best practice serverless applications in AWS [6] as the benchmarks, to investigate the impacts of cold and prewarm startup on the end-to-end latency of a function invocation. A benchmark may have several functions [41]. As for the hardware, we use one node to perform the computation and one node to generate function invocations. The benchmarks, software, and hardware setups are described in Section 8.

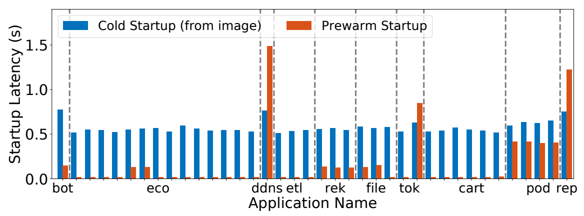


Figure 3: The cold startup latency and prewarm startup latency of the benchmarks in OpenWhisk.

Figure 3 shows the time of generating a container when it is started from the image, or is specialized from a common prewarmed container. As shown, the cold container startup takes about 500 milliseconds. The prewarm startup takes 15 milliseconds in the best case, but takes more than 1500 milliseconds in the worst case (e.g., function *union* in the benchmark *ddns*). This is because *union* requires to load/install many additional packages in the prewarmed containers, and the package loading is time-consuming.

Intuitively, a prewarmed container may install all the software packages required by all the functions on a physical node to speed up the prewarm startup. It is possible because for most serverless systems, the packages needed by a function (besides the private ones) are usually given by its user in a *requirements.txt*, and are publicly accessible for FaaS providers. However, many functions require software packages of contradicting versions. In addition, such a solution may expose the package information of other functions. The pre-imported requirements will implicitly embody user privacy. *Due to the software conflict and privacy concerns, it is not a good option to install packages for all functions in the prewarmed containers.*

3.2 Limitations of Prewarm Schemes

We then explore the effectiveness of the prewarm schemes in alleviating the cold startup. In this experiment, we run all the benchmarks on a single node, and the invocation patterns of the functions are the same as the patterns in the Azure serverless traces. The invocation patterns actually follow the Pareto distribution (most of the invocations are for a small part of the functions) [15]. By default, a prewarm container pool has two prewarmed containers on a node [2].

Figure 4 shows the percentage of the remaining cold startups with the prewarm mechanism. Many cold startups are not eliminated (e.g., functions in *eco* and *cart*). This phe-

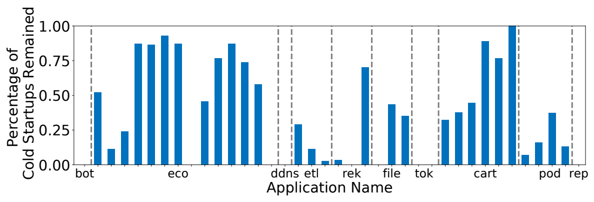


Figure 4: The remained cold startup in prewarm-enabled OpenWhisk compared with the disabled one.

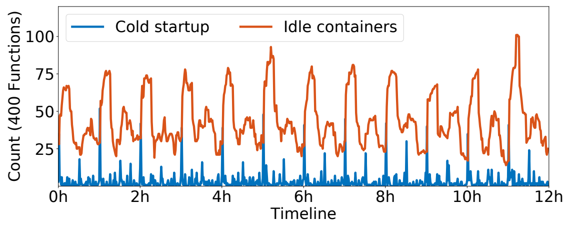


Figure 5: The number of container cold startups and idle containers from 400 randomly selected functions.

nomenon attributes to the inappropriate pool size of the prewarm container pool. For *eco* and *cart*, their functions are invoked simultaneously/or in short intervals to satisfy complex business logic, such as workflows [18, 22, 25]. For instance, five functions are triggered simultaneously by a caller in *eco*. These functions contend for the prewarmed containers.

It is nontrivial to appropriately configure the prewarm scheme due to two considerations. **1) Pool-size and memory overhead trade-off.** If we prewarm more containers, larger additional memory space is used. In our experiment, the prewarmed containers use more than 1GB of memory (on a node with 16GB memory) to eliminate 80% of the cold startup. The prewarm mechanism is not able to effectively eliminate the cold startup with reasonable memory overhead. **2) User experience and system efficiency contradiction.** As discussed, most of the invocations are from a small part of the functions [15], and a prewarmed container can only cache a small number of packages for the low memory overhead. Caching packages for frequently invoked functions improves the system efficiency (frequent invocations have low startup time), but results in poor user experience (invocations of most functions tend to suffer from the long package installation time), and vice versa.

The current container prewarm scheme is not efficient due to several inevitable trade-offs. It is beneficial to alleviate cold startups without trapping in the same dilemmas.

3.3 Opportunity of Reusing Idle Containers

We therefore propose to alleviate cold startup without relying on prewarming containers. The key idea is leveraging the warm but idle containers of some functions to alleviate the cold startups. A function invocation that requires cold startup may “steal” an idle warm container from other functions.

The proposed scheme is effective only when there are idle warm containers in some functions when an invocation tends to suffer from the cold container startup. In principle, only underutilized warm containers that are active due to the keep-alive strategy can be used by other functions. Otherwise, always stealing a warm container directly may result in the cold container startup of the victim function.

We analyze the *day*07 trace of Azure serverless platform [50] (the trace contains invocations of over 44,000 functions) to verify the above requirement. If a container triggers recycling, it must be an idle warm container because of no new invocation during its lifetime. By replaying the trace, we find that the warm containers for some functions are idle (no invocation is received during this idle time), while some other functions suffer from cold startup. We refer to idle warm containers as idle containers.

Figure 5 shows the number of idle containers and cold startups when replaying the trace. As observed, the time that idle containers and cold startup happen are similar, and there are more idle containers than the cold startups. If the time does not match, the functions that suffer from cold startup cannot find idle containers from other functions.

The time matches because many containers are prepared and invoked to serve the high load, and they become inactive when the load drops. We can observe a significant discontinuity at the beginning of each hour as there is a certain number of functions with a 1-hour timer trigger, and they are invoked once and will keep idle during the rest of their lifetime. In this case, excessive idle containers are pervasive.

In summary, serverless computing systems usually adopt a keep-alive strategy (e.g., 15 minutes) to reduce the cold startup. The kept-alive containers are idle before they are recycled. The widely-existed diurnal load pattern also makes containers over-provisioned at the high load. These containers will become idle when the load drops as well.

Based on the investigation, we observe the opportunity to leverage the idle containers of some functions to help others that suffer from cold startup on the same node.

4 Design of Pagurus

There are two prerequisites to alleviate the cold container startup with the warm containers of other functions. First, the container manager has to identify the actual idle warm containers. Otherwise, the “steal” results in the container cold startup of the victim function. Second, the proposed strategy should not expose any information of a function (e.g., data, code, package requirements) to other functions from the consideration of security.

We propose and implement **Pagurus**, a container management system that fulfills the two prerequisites. Figure 6

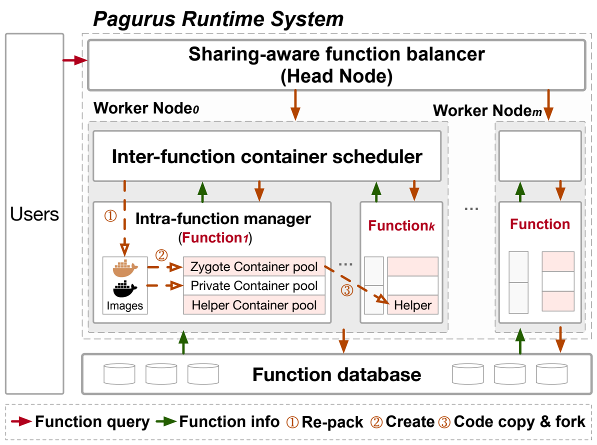


Figure 6: Design of Pagurus.

shows the design of Pagurus. It comprises an *intra-function container manager* for each function, an *inter-function container scheduler* on each node, and a *sharing-aware function balancer* at the cluster level. The intra-function container manager of a function manages its three types of containers (private containers, zygote containers, and helper containers) (Section 5). The inter-function scheduler manages the zygotes sharing between functions (Section 6). The sharing-aware function balancer maps the functions across multiple nodes to minimize the system-wide cold startup (Section 7).

Based on runtime statistics, a function's idle warm containers are replaced with its zygote containers that are newly created from its zygote image (② in Figure 6). The zygote image does not include the code or data of any functions. Pagurus uses the inter-function container scheduler on each node to generate the zygote image of each function (① in Figure 6). Creating zygote images does not introduce extra runtime latency overhead, as it is done asynchronously before replacing the idle container with a zygote container. The inter-function container scheduler determines the possible to-be-helped functions of each function based on Similarity-Filtered Weighted Random Sampling (SF-WRS) policy, which will be detailed in Section 6.1. A function's zygote container additionally installs the required packages of its to-be-helped functions in an anonymous fashion. Based on the privilege control of the Linux operating system, a function is only able to access its own packages.

Specifically, when an invocation of a function f arrives, it obtains a container to host the invocation in four steps.

1) It first tries to obtain an idle warm private container from its own private container pool directly. Then, if its private container pool is empty, it checks whether its helper container pool has containers for queries.

2) If both the private pool and the helper pool are empty, it will further check whether its zygote container pool has some containers already adapted for other functions. If not empty, a zygote container can be used to host the invocation.

3) If its zygote container pool is also empty, Pagurus tries to find a container that includes the required packages of f from other functions' zygote container pool. The forked zygote then joins the helper container pool of f (③ in Figure 6).

4) If all the above steps fail, the invocation of f would suffer from a cold container startup.

5 Intra-function Container Management

The key points of the intra-function manager are identifying the actual idle containers, and designing an efficient sharing mechanism.

5.1 Identifying Idle Containers

In principle, a container is idle when it does not host function invocations for a long time. For a function f, we introduce a timer in each of its containers to measure the free time. A warm container is treated to be idle if its timer exceeds threshold $T_{idle}(f)$. The timer is reset once the container receives an invocation.

The design principle here is that most function invocations can still get warm containers, even when a container is identified to be idle and "stolen" by other functions. Different functions should have different idle thresholds because of their diverse invocation patterns. We explore the runtime invocation arrival pattern to determine the value of $T_{idle}(f)$ for a function f. Specifically, we use all the m invocations during the container lifetime, and let T_1, T_2, ..., T_m represent the time intervals between the adjacent invocations (the time interval is sorted in the ascending order). Equation (1) calculates the idle threshold $T_{idle}(f)$ for the function f in the next time period.

$$T_{idle}(f) = \begin{cases} T_{\lceil 0.95m \rceil} & ,m \geq 30, \\ T_{default} & ,m < 30. \end{cases} \quad (1)$$

In the equation, $T_{\lceil 0.95m \rceil}$ is the 95%-ile time interval of the m samples. In this case, for frequent invoked functions ($m \geq 30$), more than 95% invocations of f tend to get warm containers, if the invocation arrival patterns remain. We use 30 to be the sampling target for stability considerations. For occasionally invoked functions, $T_{idle}(f)$ is set to be $T_{default}$, and almost all the invocations get warm containers. $T_{default}$ may impact the overall efficiency for idle identification, and Section 8.4.3 evaluates the sensitivity of Pagurus to it.

5.2 Replacing Idle Containers with Zygotes

A function's idle containers cannot be used by other functions directly, as the data and code of the function may still reside in the memory of the idle containers. To this end, Pagurus creates a zygote container that does not include any

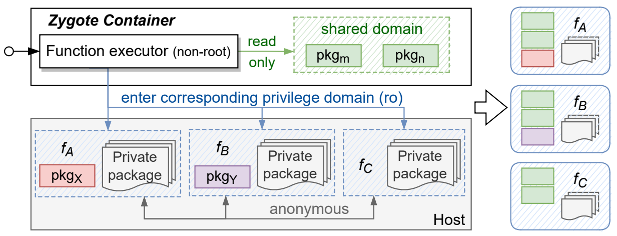


Figure 7: Security assurance of a zygote container.

data or code of the owner function, and uses the zygote container to replace the original idle warm container. The zygote container is created from an image that installs the shared packages of all the to-be-helped functions. Section 6 discusses the policies used to determine the to-be-helped functions and generate the zygote image of a function.

One may be concerned about the security and privacy of the zygote container for re-purposing. Figure 7 shows the way to avoid package information leakage in zygote containers. All the functions run as non-root users [4, 10, 16]. In the figure, f_B and f_C are the to-be-helped functions in f_A's zygote container. In general, the common intersectant packages required by all functions are installed as a shared domain (pkg_m, pkg_n) in the zygote container, and the additional complementary and private packages of to-be-helped functions are cached in different directories of the host. These directories are mounted anonymously into the zygote, thus ensuring that others cannot identify a function.

Each function that may use the zygote container is given a privilege domain and is only allowed to access its corresponding package directory. The privilege domain and privilege control are provided by Linux operating system and different non-root users. For instance, in Figure 7, when function f_B obtains the zygote container, it can only enter its own privilege domain for f_B's packages (private packages and pkg_Y) to specialize its software runtime. In this way, the zygote container serves as a safety checkpoint. Because it does not import any user-related code and data, the function privacy of the software environment is also protected.

6 Inter-function Container Scheduling

The inter-function container scheduler selects to-be-helped functions for zygotes, re-packs zygote images, and manages the fork operation for helper containers.

6.1 Selecting To-be-helped Functions

A straightforward approach is to treat all the other co-located functions as the candidate to-be-helped functions, and install all the required packages into a zygote image. However, this approach suffers from extremely high re-packing overhead, in terms of both time and resource consumption. When re-packing a zygote image, we have two important observations. On the one hand, if the set of to-be-installed packages is large, it is time-consuming and resource-unfriendly to create a giant zygote image. On the other hand, some functions tend to have more cold startups than others (as observed from Azure traces [12]), inappropriate selection of to-be-helped functions is inefficient in alleviating the system-wide cold startups. Taking the above challenges into consideration, we propose **SF-WRS** (Similarity Filtered Weighted Random Sampling) algorithm, which contains:

- A Similarity-based Filter to find out to-be-helped candidates based on the similarity of functions' packages. In this way, a zygote installs fewer complementary packages, thereby lowering the re-packing overhead.
- A WRS (Weighted Random Sampling) strategy [29] to select K to-be-helped functions based on the cold startup frequency of each function on the node. Pagurus tends to prepare zygote images for the functions that suffer from more cold startups with high possibility.

Similarity-based Filter. Focusing on the package information, a function f can be viewed as a set of packages, i.e., $f = \{pkg_1, pkg_2, ...\}$, where pkg_i is assigned in the *requirements.txt* and refers to the required package in f's runtime environment. Let $\mathbb{F}_n = \{f_1', f_2', ...\}, \forall f_i' \neq f$ represent the set of functions on node n when function f triggers re-packing. The package difference between f and $f_i' \in \mathbb{F}_n$ imposes a deep influence on the re-packing overhead. Let us denote the containment relationship of a package pkg in f_A and another function f_B as

$$con(pkg, f_A) = \begin{cases} 1 \text{ ,if } pkg \in f_A, \\ 0 \text{ ,} others, \end{cases} \forall pkg \in f_A \bigcup f_B. \tag{2}$$

We can then derive the containment relationship vector of f and $f_i' \in \mathbb{F}_n$ as $\vec{f} = \{con(pkg, f) | \forall pkg \in f \bigcup f_i'\}$ and $\vec{f_i'} = \{con(pkg, f_i') | \forall pkg \in f \bigcup f_i'\}$, respectively. The similarity between f and f_i' thus can be calculated as their cosine distance by

$$Cos(\vec{f}, \vec{f_i'}) = \begin{cases} \dfrac{\vec{f} \cdot \vec{f_i'}}{\|\vec{f}\| \|\vec{f_i'}\|} \text{ , } \|\vec{f}\| \|\vec{f_i'}\| \neq 0, \\ 1 \text{ , } \|\vec{f}\| \|\vec{f_i'}\| = 0. \end{cases} \forall f_i' \in \mathbb{F}_n. \tag{3}$$

Thereafter, we can obtain an initial f's to-be-helped function candidate set $\mathbb{C}_n^f$ by removing those functions with similarity lower than $TargetSimilarity$ from $\mathbb{F}_n$. $TargetSimilarity$ can be set as the median similarity in default.

Note that, as discussed before, a package may be specified in different versions, and a zygote image with version conflict (i.e., the same package but different versions) could not be re-purposed by another function. Denoting the version of a package pkg in function f as $V(pkg, f)$, we can express the version conflict relationship between f and f' as

$$Conflict(f, f') = \begin{cases} 1 \text{ ,} \exists \ pkg \in f, V(pkg, f) \neq V(pkg, f'), \\ 0 \text{ ,} others. \end{cases} \tag{4}$$

Then, we first ensure that there is no conflict between function f and its candidates, by removing the functions with version conflict from the candidate set. However, it is still possible that some candidates conflict with each other. For instance, candidates f'_1 and f'_2 do not conflict with f, but f'_1 conflicts with f'_2. In this case, they cannot be packed into a single zygote image either. The candidate set $\mathbb{C}_n^f$ should be further updated by Equation (5), where $\mathbb{C}_n^f \backslash f'_i$ represents the complement of f'_i in $\mathbb{C}_n^f$.

$$\begin{aligned} \mathbb{C}_n^f = \{f'_i | Conflict(f, f'_i) = 0, \\ Conflict(f'_i,\ \mathbb{C}_n^f \backslash f'_i) = 0, \forall f'_i \in \mathbb{C}_n^f, \}, \end{aligned} \tag{5}$$

Take the package conflict in AWS application benchmarks as an example, function *tcp_check_transcribe* requires the package *aws_requests_auth* with version 0.4.1, while function *ep_delivery_on_package_created* requires that with version 0.4.3. It denotes that applications have various package similarities, and two functions may rely on conflicted packages. By selecting the to-be-helped functions based on package similarity, a zygote image installs fewer packages for zygote images with less re-packing overhead.

WRS selection. After the similarity filter and conflict recognition, the to-be-helped function candidates can be significantly reduced. However, it is still nontrivial for the inter-function scheduler to determine the appropriate number of to-be-helped functions without resulting in too large image size, too long image generation time, or failing to eliminate most cold startups. Therefore, we should choose an appropriate number, say K, of functions from candidates $\mathbb{C}_n^f$ that tends to eliminate the cold startups with high probability.

Therefore, we first remove the functions never re-invoked from $\mathbb{C}_n^f$. Let I be the number of remaining functions that have been invoked more than once. We can calculate K as

$$K = \frac{\sum_{n=1}^{I} K_n}{I} = \sum_{n=1}^{I} \left[\frac{\sum_{n=1}^{I} Cold(f'_n) + \sum_{n=1}^{I} Zygote(f'_n)}{(Cold(f'_n)+Zygote(f'_n))Num(Zygote)}\right] / I, \tag{6}$$

K_i is the expected number of to-be-helped functions for f'_i, $Num(Zygote)$ is the average number of active zygotes in the system, $Zygote(f'_i)$ and $Cold(f'_i)$ indicate the times a function experiences zygote-based invocation and cold startups of f'_i, respectively. K_i ensures that each to-be-helped function f'_i can be re-packed into a zygote container at least once.

Algorithm 1 summarizes the SF-WRS algorithm. First, the inter-function scheduler filters out the candidate functions with low similarity values (lines 1-3), recognizes the package conflicts (lines 4-6), and then selects K to-be-helped ones by the A-ExpJ algorithm, which is a variation of WRS (Weighted Random Sampling) algorithm [29] (lines 9-12). Compared with naive WRS, A-ExpJ shows much lower time complexity. The time complexity of selecting K functions is $O(K \log(\frac{n}{K}))$ with A-ExpJ, and the time complexity of naive WRS is $O(n)$.

Algorithm 1 SF-WRS Selection Algorithm

```
Require: To-be-helped function candidates $\mathbb{C}_n^f$
Require: $Cold(f'_i)$ and $Zygote(f'_i)$ of function $f'_i$ in last hour
 1: $\mathbb{C}_n^f = Sample.init(\mathbb{F}_n)$
 2: for $f'_i$ in $\mathbb{C}_n^f$ do
 3:     if $Cos(\vec{f}, f'_i) < TargetSimilarity$ then: $\mathbb{C}_n^f.delete(f'_i)$
 4: for $f'_j$ in $\mathbb{C}_n^f$ do
 5:     if $Conflict(f, f'_j)$=1 or $Conflict(f'_j,\ \mathbb{C}_n^f \backslash f'_j)$=1 then
 6:         $\mathbb{C}_n^f.delete(f'_j)$
 7: if $\mathbb{C}_n^f \neq Null$ then
 8:     $Total = \sum_{n=1}^{K} Cold(f'_i) + \sum_{n=1}^{K} Zygote(f'_i)$
 9:     for $f'_k$ in $\mathbb{C}_n^f$ do
10:         $P_{repack}(f'_k) = [Cold(f'_k) + Zygote(f'_k)]/Total$
11:         $Sample.append((f'_k, P_{repack}(f'_k)))$
12:     $A-ExpJ(Sample, K)$
```

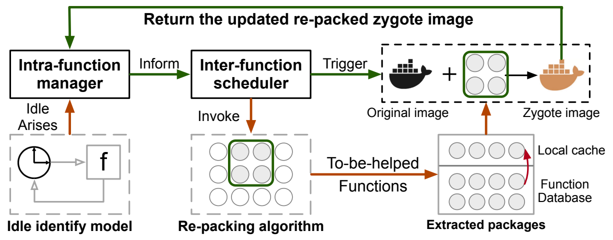


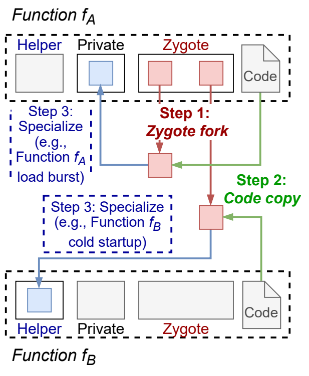
Figure 8: The key steps (represented in green) of re-packing a zygote image for the first time.

6.2 Re-packing a Zygote Image

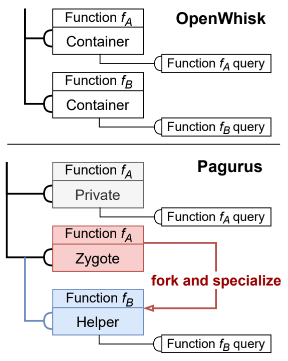
Figure 8 shows the steps of re-packing the zygote image for a function f. When the intra-function container manager of f identifies an idle container, it informs the inter-function scheduler, then selects the to-be-helped functions of f based on the SF-WRS algorithm. After that, the inter-function scheduler triggers the re-packing operation, obtains the packages, and re-packs the zygote image. Only the packages shared by all the to-be-helped functions are installed in the shared domain. Finally, the re-packed zygote image is returned to the intra-function container manager of f for building zygote containers to replace f's idle containers.

The inter-function container manager has an advantage compared with the traditional building method, where the image is built through the network from the container repository. Pagurus omits the downloading of the required packages in a zygote image through the network again, benefiting from the locally cached packages, when creating the private container images [26, 39]. By reusing cached packages, re-packing a zygote image takes a much shorter time.

Besides, the zygote image of a function is asynchronously re-packed before its to-be-helped functions actually meet in cold startups. Re-packing a zygote image does not result in long response latencies of to-be-helped function invocations.



(a) Security



(b) Process tree

Figure 9: Security guarantee when forking a zygote.

6.3 Forking a Zygote Container

The rest undertaking is to safely and efficiently share zygote containers for other functions. To provide higher availability for multiple to-be-helped functions getting zygote containers, we fork the zygote container to be helper containers, rather than directly specializing it. The zygote container remains there for other to-be-helped functions. The forked containers also ensure software security by the anonymous mounting and privilege domain as zygote containers do.

When a zygote container is forked from function f_A to be f_B's helper container, the code of f_B is copied into the forked one. We implement two plugins *Zygote fork* and *Code copy*, in the inter-function scheduler. The *Zygote fork* plugin forks a container asynchronously, un-mounts the package directories of functions other than f_B, and transfers the control access to f_B's corresponding privilege domain. The *Code copy* plugin copies the code of f_B to the helper container.

Figure 9(a) shows the steps of f_B forking a zygote container of function f_A. In Step 1, a zygote container of f_A is forked through the *Zygote fork* plugin. Then, the *Code copy* plugin copies the code of f_B into the forked zygote (Step 2). Lastly, the forked container joins the helper container pool of f_B (Step 3). It also provides process-level isolation for queries, as shown in Figure 9(b).

7 Sharing-aware Function Balancing

In existing serverless computing clusters, hash-based methods or resource usage-based methods are often used to route user queries [13, 30, 35, 37, 38, 42, 43]. It is possible that the functions on a node do not share many packages. In this case, the host node needs to create many private directories with many packages, resulting in poor resource efficiency. To address such a problem, a straightforward solution is checking the package similarities of all the active functions, and assigning the functions that share more packages to the same node. However, it is not always a good solution, as the functions sharing many packages may not have idle containers.

To resolve the problems above, we propose a function balancing strategy based on the statistics of zygote containers and the available resources $\mathbb{U}_n$=$\{U_{CPU}, U_{IO}, U_{net}, \ldots\}_n$ on every node. The function balancer is implemented on the head node of the cluster, to obtain the statistics from all the nodes. For a function f_B that requires package pkg_a and pkg_b, if it fails to find a zygote container during its invocation on a node, its future invocations should be redirected to another node with potential zygote images.

To this end, Pagurus runs the sharing-aware function balancer on the head node based on the resource usage $\mathbb{U}_n$, and the similarity between the redirected f_A and functions with idle containers on node n. Let $\mathbb{N} = \{n | \max \mathbb{U}_n \leq T_{res}\}$ represent the set of nodes where the resource utilization is under the threshold T_{res} (80% by default). The head node will select a new node with the most zygote containers from $\mathbb{N}$ and inform the API gateway accordingly. After that, the queries of f_B will be routed to this new node.

8 Evaluation of Pagurus

In this section, we evaluate Pagurus in reducing the cold startups and end-to-end latencies when a function does not have warm containers. Then, we evaluate Pagurus by a large-scale evaluation with Azure trace. After that, we show the integration with other techniques and overhead.

8.1 Experimental Setup

We use 10 best-practice applications with the most GitHub stars from Amazon AWS samples as the benchmarks [6]. We use these applications for revealing the performance of Pagurus for real applications. Experiments with small scale benchmarks in serverless benchmark suites, e.g., FaaS-Profiler [49] and ServerlessBench [61] show similar results. We run the benchmarks on a 6-node cluster. A node generates function invocations, and the other 5 nodes serve invocations. Table 1 shows the configurations of each node.

Pagurus does not rely on the function invocation arrival distribution. In Section 8.2-8.3, we send queries to each application following a Poisson distribution by randomly sampling λ between 0 and 5 queries per second. We co-locate all the benchmarks, and run 20 tests with different samples to avoid randomness. More experiments are done with the Pareto distribution-based invocation pattern of the Azure serverless trace in Section 8.4.

We compare Pagurus, prewarm-disabled OpenWhisk, prewarm-enabled OpenWhisk (OpenWhisk-Prewarm), and SOCK [46]. SOCK also prewarms containers by dynamically updating packages in the prewarmed containers to alleviate cold startups. When a function obtains a prewarmed

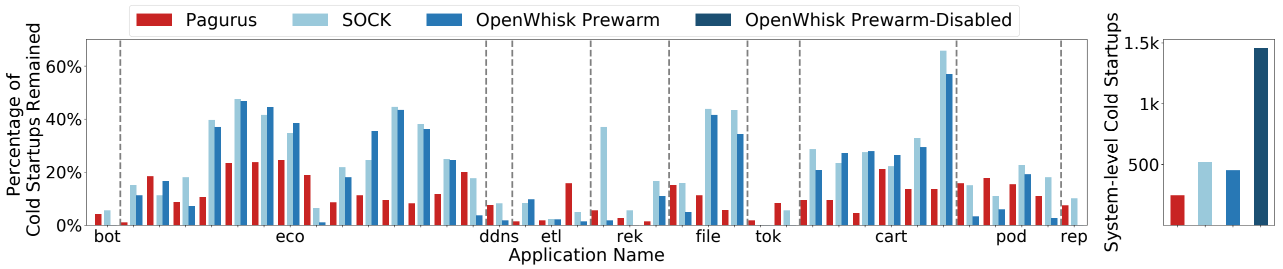


Figure 10: Remained cold startups of OpenWhisk-Prewarm, SOCK and Pagurus, compared with prewarm-disabled OpenWisk. The left figure shows the remained cold startups (all bars are normalized to the number of cold startups of OpenWhisk Prewarm-Disabled) of each function, while the right figure shows the remained cold startups of all functions in the system.

Table 1: Hardware, software, and benchmark setups

	Configuration
Node	CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz Cores: 8, DRAM: 16GB, Disk: 100GB SSD (3000 IOPS)
Software	Operating system: Linux with kernel 4.15.7, Docker: 20.10.6 Nginx version: nginx/1.10.3, Database: Couchdb:3.1.1 runc version: 1.0.0-rc93, containerd version: 1.4.4
Container	Container runtime: Python-3.7.0, Linux with kernel 4.15.7 Resource limit and Lifetime: 1-core with 256MB, 600s Function container limit: 10 for each function on each node prewarm pool size in OpenWhisk: 2 on each node
Benchmarks (38 functions in 10 AWS Lambda best practice applications)	serverless-ecommerce-platform (eco), etl-orchestrator (etl) cost-explorer-report (rep), serverless-tokenization (tok) transcribe-comprehend-podcast (pod), serverless-chatbot (bot) serverless-shopping-cart (cart), refarch-fileprocessing (file) finding-missing-persons-using-rekognition (rek), ddns

container, SOCK and OpenWhisk-Prewarm copy the missing packages into the prewarmed container for its invocation.

8.2 Alleviating Container Cold Startups

Figure 10 shows the percentages of the cold startups not eliminated by Pagurus, SOCK, and OpenWhisk-Prewarm, compared with prewarm disabled OpenWhisk. On average, Pagurus alleviate 83.1% of the cold startups, while OpenWhisk-Prewarm and SOCK alleviate 68.9% and 64.4% of that. Meanwhile, as Pagurus does not need to prewarm containers, there is no extra memory overhead introduced by the prewarm container pool with Pagurus.

As observed, SOCK and OpenWhisk-Prewarm only reduce a small percentage of cold startups for many functions (e.g., functions of *eco* and *cart*). This is because the functions tend to contend for the limited prewarmed containers. On the contrary, Pagurus alleviates the cold startups by forking other functions' zygote containers, without trapping in the same dilemmas. We can also find that Pagurus alleviates slightly fewer container cold startups for several functions, compared with SOCK and OpenWhisk-Prewarm. This is because these functions have low cold startup frequency by default, and SF-WRS policy does not tend to re-pack them into zygotes for achieving higher system-level cold startup alleviation. The percentage looks large when the original cold startup frequency is low, even if a small number of cold container startup is not eliminated.

In our experiment, 27.8% of the warm containers are turned into zygote containers, then are forked by other functions. When a function does not have a warm container for its queries, 60.5% of the obtained containers are forked.

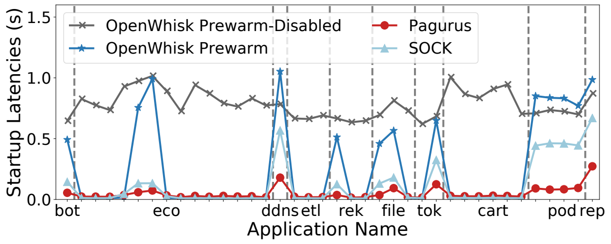


Figure 11: The latency of starting up a prewarmed container in OpenWhisk-Prewarm and SOCK, and the latency of forking a zygote container in Pagurus.

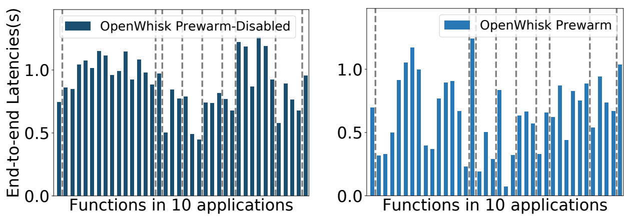
8.3 Reducing Startup and E2E Latency

Figure 11 shows the latencies of starting a container from a prewarmed one (OpenWhisk-Prewarm and SOCK), and forking a zygote container (Pagurus). As observed, all the benchmarks have the shortest startup latencies with Pagurus.

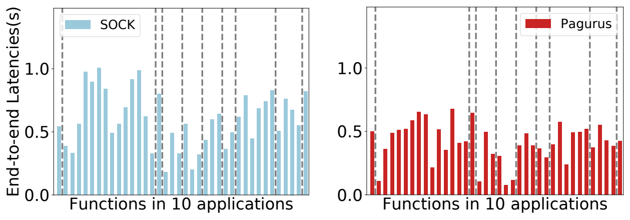
If a function invocation is hosted by a helper container with Pagurus, the packages are ready beforehand, and only the user-specific code initialization is needed. Pagurus is able to fork a zygote container in $11ms$, and completes the code initialization in $5ms$.

With OpenWhisk-Prewarm, starting from prewarmed containers takes longer than directly cold startup a container from the image (e.g., functions in *ddns*, *pod*, and *rep*). SOCK reduces the latency of the prewarm startup leveraging the packages cached with higher benefit-to-cost.

Figure 12 shows the average end-to-end latency of each function normalized to its latency with prewarm-disabled



(a) OpenWhisk Prewarm-Disabled (b) OpenWhisk-Prewarm



(c) Sock (d) Pagurus

Figure 12: The end-to-end latencies of Prewarm-Disabled OpenWhisk, OpenWhisk Prewarm, SOCK and Pagurus.

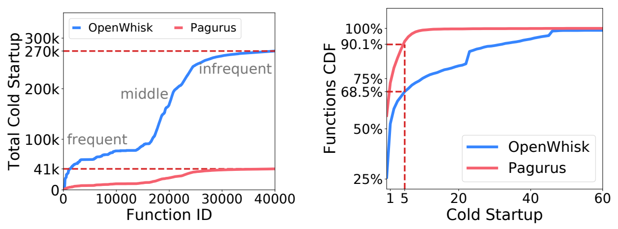
OpenWhisk. Pagurus reduces the end-to-end latency of the benchmark functions by 475ms and 479ms on average, while OpenWhisk-Prewarm and SOCK reduce the end-to-end latency by 237ms and 286ms.

By mounting packages beforehand with privilege control, zygote containers can better reduce the startup latency, thus the end-to-end latency.

8.4 Large-scale Evaluation with Azure Trace

In this subsection, we evaluate Pagurus by replaying the Azure serverless trace [50] on a 31-node cluster. The software and hardware configuration of each node is the same as Table 1. We use all the 40,000 functions from the *day*07 trace [12], generate function invocations, and randomly route the invocations to the 30 nodes.

The Package similarity in Pagurus is used to shrink the searching space for identifying to-be-helped candidates and reducing the re-packing overhead. However, the Azure trace does not provide package information for the functions, but only the function duration and invocation arrival time. Lacking the package information, it is impossible to evaluate the similarity-filtered WRS selection policy in Pagurus, nor OpenWhisk-Prewarm or SOCK. With no package information and similarity-filtered re-packing for Azure trace, Pagurus identifies to-be-helped candidates based on the basic WRS policy. We therefore only compare similarity-disabled Pagurus, with the prewarm-disabled OpenWhisk for the large-scale evaluation , to show the effectiveness of alleviating cold startups by simply replacing idle containers with zygotes.



(a) Cumulative cold startups (b) Frequencies of cold startups

Figure 13: The effect of alleviating cold startup, and the CDF that functions suffer from different cold startup frequencies.

8.4.1 Alleviating Cold Startup

We report two user experience-related metrics in this experiment. First, how many cold startups are alleviated by Pagurus? Second, how many functions seldom experience cold startups (e.g., one cold startup) in one day with Pagurus?

Figure 13(a) shows the total number of cold startups with Pagurus and prewarm-disabled OpenWhisk (denoted by "OpenWhisk" for short in this subsection). In the figure, the functions are sorted in the descending order of their invocation frequencies. The smaller the function ID, the more frequent the function is invoked. As shown in the figure, Pagurus reduces the number of cold startups by 84.6%. We can also find that OpenWhisk results in the frequent cold startup for the functions of middle-popularity. It is because the warm containers of these middle-popularity functions tend to be recycled due to the relatively low invocation frequencies. Pagurus efficiently alleviates the cold startups for the middle-popularity functions through zygote containers.

Figure 13(b) shows the cumulative distribution of the functions with different container cold startup frequencies. As observed, 73.4% and 52.1% of all functions experience cold startup less than once in a day with Pagurus and OpenWhisk, respectively. Meanwhile, 90.1% of the functions experience cold startups less than 5 times daily with Pagurus. In the figure, sudden jumps happen around 24 and 48 cold startups for OpenWhisk. The jumps are caused by functions with a 1-hour or 30-minutes trigger in the trace.

Pagurus effectively alleviates the cold container startup, especially for middle-popularity and low-popularity functions in production. It greatly improves the user experience.

8.4.2 Reducing Tail Latency

Figure 14 shows the 95%-ile latencies of the 40,000 functions with Pagurus and OpenWhisk. The left y-axis shows the 95%-ile latencies of functions with Pagurus, and the right y-axis shows that with OpenWhisk normalized to the former. OpenWhisk results in longer 95%-ile latencies for most functions than Pagurus (the right y-axis is larger than 1).

We can also observe that popular functions (functions

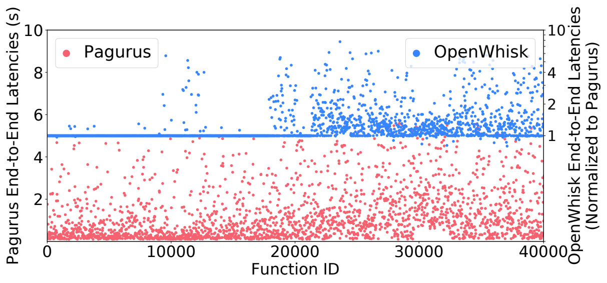


Figure 14: The end-to-end 95%-ile latency of 40,000 functions with Pagurus and OpenWhisk.

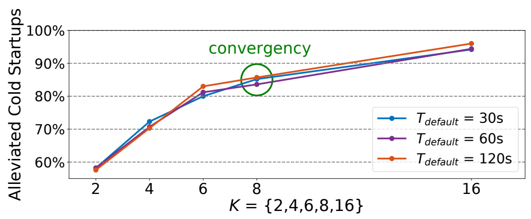


Figure 15: The convergency and the percentages of cold startups alleviated by Pagurus with different $T_{default}$.

with ID smaller than 15,000) have similar 95%-ile latency with OpenWhisk and Pagurus. This is because the slowest 95%-ile invocations of these functions still experience warm startup, as these functions are frequently invoked. For the middle-popularity and low-popularity functions, their 95%-ile latencies are the latency of the function invocation that suffers from the cold container startup with OpenWhisk.

8.4.3 Impacts of Hyperparameters

In this experiment, we evaluate the impact of $T_{default}$ (the value is set as 60s by default), and the number of to-be-helped functions K, on Pagurus. Figure 15 shows the percentages of cold startups alleviated by Pagurus with different $T_{default}$ and different K. As observed, the performance of Pagurus is stable when $T_{default}$ varies. The performance of Pagurus is not sensitive to $T_{default}$.

We also find the number of to-be-helped functions, K, gradually converges to 8. Moreover, the appropriate value of K is not affected by $T_{default}$. For Azure workloads, Pagurus should generate a zygote for 8 to-be-helped functions on average. It is consistent with the calculated one in Equation 6. We also measure the impact of larger prewarm pool size in OpenWhisk, and find that the improvement becomes marginal but with significant resource waste.

8.5 Integrating with Orthogonal Techniques

The decoupled hierarchy design of Pagurus provides easy-to-use APIs for container orchestrators. Pagurus can be integrated with prior work on speeding up the cold startup.

Table 2: Overheads of the components in Pagurus

Sources	Type	Overheads (each node)
Intra-container manager	CPU overhead	0.345 core
	Memory overhead	228MB
	Storage overhead	485MB for each zygote image
Inter-function scheduler	CPU overhead	0.66 core (re-packing included)
	Memory overhead	315MB

Pagurus brings shorter end-to-end latency when it is integrated with Checkpoint/Restore [7] (denoted by C/R) and Catalyzer [28], respectively. With C/R, a container is recovered from a checkpoint image. With Catalyzer, more data are already loaded in the image stored in memory. By replaying the evaluation, we find that C/R+Pagurus reduces the cold startup time of the benchmarks by 78.9% on average compared with C/R; Catalyzer+Pagurus reduces the cold startup time by 15.1% on average compared with Catalyzer. Even if no appropriate forked zygote container returns, Pagurus does not slow down the container startup.

8.6 Overheads of Pagurus Components

In Pagurus, *packing zygote images*, *generating zygote containers from the images*, and *determining the to-be-helped functions* for each function introduce runtime overhead.

According to our measurement, each container in Pagurus uses smaller memory on average than OpenWhisk. The reduction originates from the design of the zygote container. Although packages are pre-installed in zygote containers, they are imported into memory only when a zygote container is forked. On the contrary, warm containers (private containers) always keep the packages in memory for low latency. The reduction of memory usage is not affected by the number of to-be-helped functions.

Table 2 shows the CPU, memory, and storage overhead caused by the intra-container managers and the inter-function schedulers when replaying the Azure trace. As reported, less than one core is required to run all the intra-container managers and the inter-function scheduler on a node. If fewer functions are executed on a node, the overhead will be smaller.

9 Conclusion

Pagurus alleviates cold startups with inter-function container sharing rather than popular prewarm-based methods. It comprises an intra-function manager for idle container identification and management, an inter-function scheduler for safe container scheduling, and a sharing-aware function balancer for resource-aware workload balancing. Our experimental results based on both real system benchmarks and Azure trace show that Pagurus significantly alleviates the cold container startup. The cold startup latency is reduced from hundreds of milliseconds to 16*ms* if Pagurus alleviates it.

Acknowledgment

We would thank our anonymous reviewers and shepherd, for their helpful comments and suggestions. This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61872240), Open Research Projects of Zhejiang Lab (2021KE0AB02), and Shanghai international science and technology collaboration project (21510713600). Quan Chen and Minyi Guo are the corresponding authors.

References

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

[2] Annotations on openwhisk assets. https://github.com/apache/openwhisk/blob/90c20a847b9a70b43e316fd89a0a15ae2ee39cc4/docs/annotations.md, 2021.

[3] Apache openwhisk. https://openwhisk.apache.org, 2021.

[4] Authentication and authorization in azure functions. https://docs.microsoft.com/en-us/azure/app-service/overview-authentication-authorization, 2021.

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

[6] Aws samples. https://github.com/aws-samples/, 2021.

[7] Checkpoint/restore. https://criu.org/Checkpoint/Restore, 2021.

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

[9] Fission workflows: Fast, reliable and lightweight function composition for serverless functions. https://docs.fission.io/docs/workflows/, 2021.

[10] Function identity in google cloud functions. https://cloud.google.com/functions/docs/securing/function-identity, 2021.

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

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

[13] Nginx. https://www.nginx.com/, 2021.

[14] Prewarm in apache openwhisk. https://github.com/apache/openwhisk/blob/master/docs/actions-python.md, 2021.

[15] Prewarm in azure functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-premium-plan, 2021.

[16] Troubleshooting aws lambda identity and access. https://docs.aws.amazon.com/lambda/latest/dg/security_iam_troubleshoot.html, 2021.

[17] RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.

[18] Mainak Adhikari, Tarachand Amgoth, and Satish Narayana Srirama. A survey on scheduling strategies for workflows in cloud environment and emerging trends. *ACM Comput. Surv.*, 52(4):68:1–68:36, 2019.

[19] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, and Klaus Satzke. SAND: Towards high-performance serverless computing. In *ATC*, pages 923–935, 2018.

[20] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, and Dimitris Skourtis. Improving docker registry design based on production workload analysis. In Nitin Agrawal and Raju Rangaswami, editors, *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, pages 265–278. USENIX Association, 2018.

[21] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. In Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya, editors, *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[22] Kahina Bessai, Samir Youcef, Ammar Oulamara, Claude Godart, and Selmin Nurcan. Bi-criteria workflow tasks allocation and scheduling in cloud computing environments. In *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, pages 638–645. IEEE Computer Society, 2012.

[23] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In Haryadi S. Gunawi and Benjamin Reed,

editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 645–650. USENIX Association, 2018.

[24] Eric A. Brewer. Kubernetes and the path to cloud native. In Shahram Ghandeharizadeh, Sumita Barahmand, Magdalena Balazinska, and Michael J. Freedman, editors, *SoCC*, page 167. ACM, 2015.

[25] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25(6):599–616, 2009.

[26] James Cadden, Thomas Unger, Yara Awad, and Han Dong. SEUSS: skip redundant paths to make serverless fast. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 32:1–32:15. ACM, 2020.

[27] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In Dilma Da Silva and Rüdiger Kapitza, editors, *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, pages 356–370. ACM, 2020.

[28] 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.

[29] Pavlos S. Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. *Inf. Process. Lett.*, 97(5):181–185, 2006.

[30] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, Xin Peng, Wenli Zheng, and Minyi Guo. Qos-aware and resource efficient microservice deployment in cloud-edge continuum. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*, pages 932–941. IEEE, 2021.

[31] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In Tim Sherwood, Emery 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 386–400. ACM, 2021.

[32] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In Angela Demke Brown and Florentina I. Popovici, editors, *FAST*, pages 181–195. USENIX Association, 2016.

[33] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, and Johann Schleier-Smith. Serverless computing: One step forward, two steps back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[34] 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.

[35] M. Reza HoseinyFarahabady, Albert Y. Zomaya, and Zahir Tari. A model predictive controller for managing qos enforcements and microarchitecture-level interferences in a lambda platform. *IEEE Trans. Parallel Distributed Syst.*, 29(7):1442–1455, 2018.

[36] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: a berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[37] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 158–164. ACM, 2019.

[38] Young Ki Kim, M. Reza HoseinyFarahabady, Young Choon Lee, and Albert Y. Zomaya. Automated fine-grained CPU cap control in serverless computing platform. *IEEE Trans. Parallel Distributed Syst.*, 31(10):2289–2301, 2020.

[39] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. DADI: block-level image service for agile and elastic application deployment. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 727–740. USENIX Association, 2020.

[40] 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.

[41] 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.

[42] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Marin Litoiu. Optimizing serverless computing: introducing an adaptive function placement algorithm. In Tima Pakfetrat, Guy-Vincent Jourdan, Kostas Kontogiannis, and Robert F. Enenkel, editors, *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON 2019, Markham, Ontario, Canada, November 4-6, 2019*, pages 203–213. ACM, 2019.

[43] Sean McDaniel, Stephen Herbein, and Michela Taufer. A two-tiered approach to I/O quality of service in docker containers. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, pages 490–491. IEEE Computer Society, 2015.

[44] M. Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino, editors, *ICDCS Workshop*, pages 405–410. IEEE Computer Society, 2017.

[45] Anup Mohan, Harshad S. Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 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.

[46] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with serverless-optimized containers. In *{USENIX} Annual Technical Conference (ATC)*, pages 57–70, 2018.

[47] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In Jay R. Lorch and Minlan Yu, editors, *NSDI*, pages 193–206. USENIX Association, 2019.

[48] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. 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 753–767. ACM, 2022.

[49] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Micro*, pages 1063–1075. ACM, 2019.

[50] 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, *ATC*, pages 205–218. USENIX Association, 2020.

[51] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *CoRR*, abs/1810.09679, 2018.

[52] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *ASPLOS*, pages 121–135, 2019.

[53] Eli Tilevich and Hanspeter Mössenböck, editors. *International Conference on Managed Languages & Runtimes*. ACM, 2018.

[54] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Tim Sherwood, Emery 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.

[55] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. Fast in-memory criu for docker containers. In *International Symposium on Memory Systems*, pages 53–65, New York, NY, USA, 2019. Association for Computing Machinery.

[56] 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, *SOSP*, pages 148–162. ACM, 2005.

[57] T. Wagner. Understanding container reuse in aws lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, 2021.

[58] 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, *Eurosys*, pages 39:1–39:16. ACM, 2019.

[59] 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.

[60] 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.

[61] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 30–44. ACM, 2020.

A Artifact Appendix

A.1 Abstract

Our artifact includes the prototype implementation of Zygote-based mechanism in Pagurus, 10 masked applications (including 38 functions) benchmarks, and mapped functions from Azure Trace. The artifact provides experiment workflow scripts to perform the measurement.

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

- **Run-time Environment:** Ubuntu 18.04, Docker 20.10.6, CouchDB 3.2.2 and Python are required.
- **Data set:** The artifact uses 10 masked application benchmarks from AWS samples and Azure traces.
- **Execution workflows:** For reproducing our paper's results, we provide the corresponding scripts for each evaluation section (from Section 8.2 to Section 8.4) to send queries, collect the execution metrics, and draw the comparison plots.
- **Time needed to complete:** see the instruction of each Exp.
- **Publicly available:** https://github.com/lzjzx1122/Pagurus
- **Code Licenses:** Apache-2.0 license

A.3 Hardware and Software Dependencies

- **Hardware:** The hardware is configured by {CPU: Intel Xeon(Ice Lake) Platinum 8369B @3.5GHz, Cores: 8, DRAM: 16GB, Disk: 200GB SSD with 4200 IOPS.}
- **Software environment:** Operating system: {Linux with kernel 4.15.0, Docker: {20.10.15}, Container runtime: {Python-3.6.9, Linux with kernel 4.15.7}, Nginx version: {nginx/1.10.3}, Database: {Couchdb with version 3.2.2}, runc version: {1.0.0-rc93}, containerd version: {1.4.4}, and Pagurus. Detailed software dependencies are all listed and scripted in the artifact.

A.4 How to Access and Install

GitHub link: https://github.com/lzjzx1122/Pagurus. Clone the GitHub repository and then run the quick setup script to deploy Pagurus.

A.5 Experiment and Expected Results

A.5.1 AWS applications (Section 8.2 and 8.3)

Under the path `Pagurus/aws/trace`, there are 18 different sampling test results for AWS applications, and the expected test results for each sampling are stored in the directory `aws/expected_result`. You can directly run `aws/plot.py` to generate the plots, or replay the trace using a testing script to run the AWS experiment:

```
$ python3 aws/run_experients.py 1
```

Experiment customization: The above script performs the 1st sampling test with Openwhisk, Pagurus, OpenWhisk-Prewarm and SOCK. Other sampling tests can also be performed by changing "1" to other test numbers. To fully reproduce our result, it additionally takes at least 160-hours (20 tests with different sampling parameters) to generate the execution logs for 4 platforms. To ensure that the zygote repacking mechanism and prewarm works efficiently, the running time should be at least 2-hours for both 4 platforms (8 hours for each test number).

After running the sampling tests under four platforms, the script will generate the results under `aws/result`. Run the following script to generate three `.csv` files:

```
$ python3 aws/summary_from_results.py
```

- `cold_start.csv` shows the remained cold startups of OpenWhisk, SOCK and Pagurus, compared with prewarm-disabled OpenWhisk (Figure 11).
- `startup_time.csv` shows latencies of starting up containers in prewarm-disabled OpenWhisk, OpenWhisk and SOCK, respectively. It also contains latencies of forking zygote containers in Pagurus (Figure 12).

- `e2e_latency.csv` shows e2e latencies of benchmarks in Pagurus, SOCK, OpenWhisk, and prewarm-disabled OpenWhisk, respectively. (Figure 13).

Then run the following script to generate the plots:

```
$ python3 aws/plot_from_results.py
```

A.5.2 Azure Trace mapping (Section 8.4)

In the Azure trace experiment (Day07), invoke more than 40,000 functions will take 24 hours and more than 800 vCPUs by default. To reduce the computation resources needed, we randomly select about 4,000 functions to generate several small-scale Azure traces under the path `Pagurus/azure/trace`. The functions in each small-scale trace are all different from each other. A larger trace with more functions can be replayed if only more computation resources or nodes are provided.

Considering that the experiment will take about 24 hours, we already pre-run each small-scale Azure trace and save their execution results. You can directly run `azure/plot.py` to generate the plots, or replay the trace using the following script:

```
$ python3 azure/run_experients.py 1
```

Experiment customization: Each small-scale Azure trace can be replayed by changing "1" to other trace numbers. The script will replay the trace in Openwhisk and Pagurus, respectively, and then generate results under `azure/result`. Run the following script to generate two `.csv` files:

```
$ python3 azure/summary_from_results.py
```

- `cold_start.csv` shows the remained cold startups of OpenWhisk and Pagurus, respectively (Figure 14).
- `e2e_latency.csv` shows end-to-end 95%-ile latencies of benchmarks in Openwhisk and Pagurus, respectively (Figure 15).

Then run the following script to generate the plots:

```
$ python3 azure/plot_from_results.py
```