FaaSImage: An Efficient Image Manager for FaaS

Abhisek Panda
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India
abhisek.panda@cse.iitd.ac.in

Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India
srsarangi@cse.iitd.ac.in

Abstract

The cold start latency in serverless systems is a matter of great concern. It militates against its basic foundation, which is fast millisecond-level execution of mostly stateless functions. Over the last five years, a lot of work has been done in academia and industry to mitigate the overheads caused by long cold start times. In this paper, we focus on a specific line of work that proposes to modify the Docker container's architecture to address this problem. We observe that state-of-the-art work has either fused Docker layers or used smart on-demand fetching of data.

We take a deep look at this problem and study the behavior of Docker containers in serverless setups in great detail. We observe that different containers access different subsets of image files, and there is a fair amount of variance across applications. A container typically accesses 30% of the lower layers that store OS and runtime data, and at most 69% of the application layers that store code, libraries and dependencies. Furthermore, there are varying degrees of overlap between layers of different containers. Our proposed method FaaSImage is the most *comprehensive proposal* in this space till date – it combines three different techniques, namely layer subsetting, on-demand download and layer fusion. Furthermore, it uses different strategies to pull different types of layers. FaaSImage reduces the image pull latency by at most 52% on HDDs and 58% on SSDs compared to state-of-the-art solutions. Upon integration with OpenFaaS, we observe that the cold start latency of functions reduces by 38.7% on HDDs and 39.8% on SSDs.

CCS Concepts

• **Computer systems organization** → **Cloud computing**; • **Information systems** → **Cloud based storage**; • **General and reference** → *Performance*.

Keywords

Serverless computing, Function-as-a-Service, Container Deployment, Docker Image Format

ACM Reference Format:
Abhisek Panda and Smruti R. Sarangi. 2025. FaaSImage: An Efficient Image Manager for FaaS. In *26th ACM Middleware Conference (Middleware '25), December 15–19, 2025, Nashville, TN, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721462.3730951

1 Introduction

Serverless computing is a very popular cloud computing paradigm in which the responsibility of application deployment and resource management shifts from developers to cloud platforms [26, 27]. Major commercial serverless platforms such as Microsoft Azure Functions [7], Google Cloud Functions [10], IBM Cloud Functions [11] and Amazon Lambda [32] provide autoscaling features, fine-grained billing models and fault tolerance services to developers. Autoscaling features dynamically regulate resource allocation for an application based on incoming traffic. Billing models charge developers for the duration of the application's execution only. Because of these features, developers can focus solely on the development process and delivering value to customers. Hence, in recent years, there has been a significant increase in the adoption of this paradigm by many applications in fields such as IoT, machine learning, data analytics and large cloud-based portals such as Netflix, FINRA, iRobot [3, 12, 28].

A user uploads applications to a serverless platform as a set of functions along with their corresponding execution sequence, unlike other cloud computing platforms. Subsequently, the platform handles application requests by treating them as a sequence of function requests. For every request, the platform first *pulls* a function image from a remote registry that contains the operating system files, the runtime environment, the function code, dependencies and libraries. An *image* can be either a single file or a stack of *read-only* layers (stored as a tar file), where each layer represents a portion of the image. Subsequently, the platform creates a sandbox using the image and executes the function within it. Therefore, the cost of pulling an image (subject to its availability) and creating a sandbox are key determinants of a *function's cold start latency* [26, 27, 29, 43, 56].

To improve a function's cold start latency, prior work has focused on the following techniques: sharing a sandbox [2], utilizing lightweight sandbox mechanisms [1] and restoring from a sandbox's stored snapshot [5, 47]. However, these techniques improve the cost of sandbox creation and assume that images are already present on a machine's local disk, which may seldom be true [57]. A recent study by Microsoft has shown that approximately 50% of serverless workloads exhibit substantial variation in the inter-arrival times of requests [46]. Consequently, the probability of a function image being available on a node is relatively low (owing to replacements).

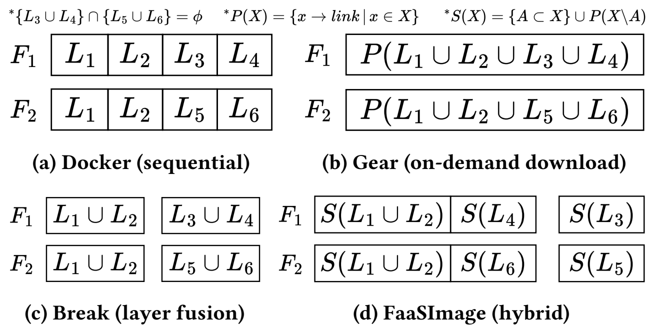

$^*\{L_3 \cup L_4\} \cap \{L_5 \cup L_6\} = \phi$
$^*P(X) = \{x \to link \mid x \in X\}$
$^*S(X) = \{A \subset X\} \cup P(X \setminus A)$
F_1 L_1 L_2 L_3 L_4
F_2 L_1 L_2 L_5 L_6
(a) Docker (sequential)
F_1 $P(L_1 \cup L_2 \cup L_3 \cup L_4)$
F_2 $P(L_1 \cup L_2 \cup L_5 \cup L_6)$
(b) Gear (on-demand download)
F_1 $L_1 \cup L_2$ $L_3 \cup L_4$
F_2 $L_1 \cup L_2$ $L_5 \cup L_6$
(c) Break (layer fusion)
F_1 $S(L_1 \cup L_2)$ $S(L_4)$ $S(L_3)$
F_2 $S(L_1 \cup L_2)$ $S(L_6)$ $S(L_5)$
(d) FaaSImage (hybrid)

Figure 1: Layer organization and pull operation of function images: F_1 and F_2 across Docker, Gear [18], Break [19] and FaaSImage. L_1 and L_2 denote the OS and runtime; $\{L_3, L_5\}$ denote libraries; $\{L_4, L_6\}$ denote code and dependencies; and A denotes the required files for container startup. *Note: The space between layers denote the parallel fetching of layers.*

Experiments on Alibaba Cloud have also shown that platforms observe sudden spikes in a function's traffic pattern [56]. This creates network hotspots that adversely affect the pull latencies. According to Wang et al. [56] (Alibaba Cloud), approximately 86% of image pull operations in the Shanghai region take at least 80 seconds, which is very high for serverless systems.

To address this problem, the following techniques have been proposed in prior work: *caching of sandboxes and images* [8, 9, 20, 31, 35, 40, 44, 50, 52, 57, 58], *peer-to-peer (P2P) transmission of images* [17, 19, 21, 30, 56, 60], and *pre-warming of sandboxes* [37, 42, 44, 46, 61]. These techniques have matured over time, and in the humble view of the authors diminishing returns have set in. In this paper, we explore the next logical frontier that is getting good traction – Docker image engineering [18, 19, 23, 34] – in the context of serverless computing redesigning the Docker images to enable better performance in sandbox deployments. This approach can be viewed to be mostly complementary vis-à-vis prior work; it can be used along with them. In this paper, we propose *FaaSImage* as a means to improve the latency of the cold start operation while also minimizing the network bandwidth utilization. We can also seamlessly integrate it with existing serverless platforms.

Given that an image is organized as a stack of layers, the image pull pipeline sequentially extracts compressed layers that were downloaded in parallel (see Figure 1a). The image engineering mechanisms have utilized either *layer fusion* or *on-demand download* techniques. Layer fusion techniques rearrange the layers of an image to maximize the overlapping of layers across images and pull layers in parallel (see Figure 1c) [19, 34]. As a result, these techniques reduce the number of download and extraction operations during the pull operation. However, in the case where a sandbox accesses only a fraction of an image, retrieving the entire image results in resource wastage and higher pull latency. To address this, we have on-demand download techniques that pull and mount a file to a sandbox when it is accessed for the first time – this reduces the image pull latency (see Figure 1b) [18, 23].

In this paper, we propose a paradigmatically different approach – combine layer fusion and on-demand download approaches and do more (*hybrid* approach in Figure 1d). Our key observation is that a function's sandbox only accesses a fraction of the image during its startup. Therefore, it is not necessary to pull all the files of a function image from the registry. Instead, we can choose to pull some files of the image that are required for startup and retrieve the remaining files as needed during execution (*layer subsetting* + *on-demand download*). We propose a novel rearrangement of an image's files into three layers – *base*, $func$, and lib – to efficiently utilize network bandwidth and reduce the pull latency (*layer fusion*) (see Figure 1d). The *base* layer stores the operating system files and the runtime environment – common among functions using the same runtime environment and fetched only once. The $func$ layer stores the function code and dependencies – unique across functions. The lib layer stores the libraries – shared among functions. During the image pull operation, we pull the $func$ and lib layers in parallel. In the case of the lib layer, we only fetch the library's files that are missing on a machine; this further reduces the pull latency and lowers the bandwidth requirement.

To summarize, our contributions are as follows:

(1) We perform an extensive analysis of Docker's image pull pipeline and function images to identify bottlenecks during the image pull and container deployment operations.
(2) FaaSImage packages only the files necessary for startup in an image, while the remaining are fetched on-demand during execution.
(3) FaaSImage partitions an image into a common *base* layer, a unique *func* layer, and a shared *lib* layer to improve the cold start and bandwidth requirements.
(4) Upon integration with OpenFaaS, FaaSImage improves the cold start latency by 38.7% on HDDs and 39.8% on SSDs, respectively, when compared to OpenFaaS.
(5) FaaSImage improves the pull latency by at most 52% on HDDs and 58% on SSDs, respectively, when compared to other state-of-the-art techniques.
(6) FaaSImage reduces the download traffic and storage size by at most 45% compared to other state-of-the-art techniques.

We organize the rest of the paper as follows. We provide the relevant background on Docker images in Section 2. Subsequently, we perform a comprehensive analysis of the pull and deployment operations of images in Section 3. Following this, we discuss the design of FaaSImage in Section 4. Section 5 evaluates the storage performance, bandwidth utilization, and deployment latencies of FaaSImage vis-à-vis state-of-the-art solutions. We discuss the related work in Section 6 and finally conclude in Section 7. The source code is available at https://github.com/catchabhisek/FaaSImage-Artifact.

2 Background

2.1 Serverless Computing

In the serverless computing paradigm, a platform deploy applications at the granularity of functions. Users decompose an application into a set of functions and upload it to the platform along with the execution sequence. To execute a function, the platform first fetches the function image, which includes an operating system (OS), a runtime environment, the function code, and libraries. Subsequently, the platform spawns a sandbox with the function image, which involves booting the OS, initializing the runtime environment, and loading the function code and libraries (referred to as

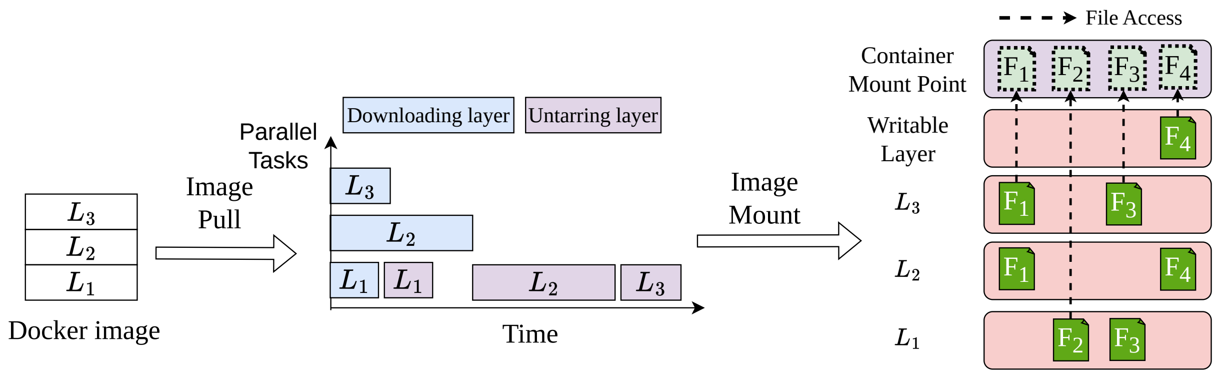


Figure 2: A system pulls a Docker image from a remote registry. Subsequently, a container uses the mount point to access files. On every access, the overlay2 storage driver searches for files in the writable layer and image (a stack of read-only layers).

the startup phase in the paper). Finally, the sandbox executes the function's request and sends the response to the platform (referred to as the execution phase in the paper). In this paradigm, platforms manage the execution of an application by using sandbox environments, which consequently reduces the operational complexity at the developer's end. Furthermore, platforms charge users based on the duration of function execution, providing a fine-grained billing scheme.

In contrast to other cloud platforms, serverless platforms spawn a sandbox prior to the execution of a function that is on the critical path. This operation results in a performance overhead, referred to as *cold start*. To mitigate the cold start problem, prior work has used the following techniques: warm containers [31, 52], and snapshot-restore [5, 47]. In warm containers, we retain a sandbox post-function execution for a specific duration of time, in anticipation of future requests. However, this technique results in the inefficient utilization of system resources. On the other hand, snapshot-restore techniques take a snapshot of the sandbox after function initialization. Subsequently, they restore a sandbox using its snapshot, thereby eliding the cost of operating system initialization, runtime initialization, and library loading. However, these techniques assume that the function image is available on the machine, thereby not showing the cost of image pull [57].

Popular serverless platforms like Apache Openwhisk [6], IBM cloud functions [11], OpenLambda [24] and OpenFaaS [36] use Docker to create sandboxes. In this paper, we use Docker to manage our containers. Let us discuss the image pull pipeline in Docker.

2.2 Docker

As the deployment of cloud-based applications becomes increasingly prevalent, service providers require a lightweight virtualization mechanism. Docker containers are a viable solution because they offer process-based isolation instead of VM-based isolation. To ensure security and performance isolation, Docker utilizes the *cgroup* and *namespace* features provided by Linux. Let us elaborate.

2.2.1 Docker Image. To create a Docker container, we first need to define the *root* filesystem's image of the container. Docker organizes an image as a stack of *read-only layers*, with each layer representing a portion of the filesystem (see Figure 2). If multiple images share a common set of base layers, we fetch the common layers only once. Additionally, when we update an image, we push a layer to the stack that reflects only the deleted, modified, or added files. This representation enhances storage efficiency, image creation, and image revision operations. For accessibility, we upload the image's layers to a centralized Docker registry that stores unique layers across images.

2.2.2 Docker Image Pull. To pull a Docker image, Docker sends a request to the registry with the image ID. It receives a `manifest.json` file containing the image's layer IDs and their sequence required to build the filesystem. Docker then performs a lookup operation on the local layer cache using layer IDs to check their availability on the machine. If a layer is absent, it fetches the compressed layer from the registry and then performs the untar operation on the compressed layer (see Figure 2). Docker uses a configurable number of layer downloader threads that are responsible for fetching compressed layers in parallel. However, the untar operation happens in a sequential manner due to its potential incompatibility with some storage drivers and its higher utilization of system resources [48].

2.2.3 Docker Storage Drivers. Docker employs various storage drivers, such as *overlay2*, *AUFS* (Another Union File System), *Btrfs* (B-Tree File System), and *ZFS* (Zettabyte File System) to superimpose the layers of an image and thus provide a *merged* view of the filesystem. The AUFS and overlay2 storage drivers operate at the file level, while Btrfs and ZFS operate at the block level [16]. ZFS and Btrfs require a lot of memory to store Docker images, which may lead to memory pressure in serverless platforms [16, 41]. We observe that during the deployment of function containers, ZFS requires an additional 856 *MB* of memory compared to the overlay2 storage driver. At the moment, Linux does not have native support for AUFS [13]. Therefore, we use the overlay2 storage driver in this paper.

2.2.4 Overlay2 Storage Driver. During container creation, Docker must provide the filesystem image's mount point to a container. But Docker stores the image as a stack of read-only layers instead of a single directory. To resolve this, Docker employs the *overlay2* storage driver that uses the overlay kernel module to superimpose the read-only layers of an image, along with a writable layer (see Figure 2). This is because the image's layers are read-only; a container stores the modified or additional files in the writable layer. When accessing a file, the overlay kernel driver first invokes `ovl_lookup()` to get the file descriptor and the inode number. Given that it superimposes multiple layers, the lookup operation must be executed on each layer to find the location of a file (referred to as `ovl_lookup_single()`).

3 Characterization of a Function Image

In this section, we evaluate OpenFaaS (an open source serverless platform [36]) and investigate the impact of a function's image on its performance on HDDs. Furthermore, we evaluate the efficacy of state-of-the-art image engineering methods used to improve the cold start operation: Break [19] and Gear [18] when integrated with OpenFaaS. For this study, we take a set of real-world serverless functions (see Table 1) adopted from FunctionBench [28]. We use a system whose configuration is shown in Table 2. The image pull latency is dependent on the Docker registry's load and the hop count from the system to the Docker registry. To have a controlled environment, we deploy a local registry on a storage server that is interconnected to the system (similar to Gear [18]). While downloading an image from the registry, we set the number of *layer downloader threads* to 3 and use *overlay2* as the storage driver (default values as per the Docker documentation) [15, 16].

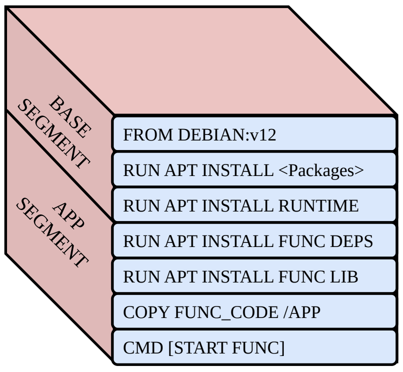


Figure 3: The typical structure of a function image. It consists of two parts: the *base segment*, which denotes a set of layers that contains OS and the runtime environment, and the *app segment*, which denotes a set of layers that contains the code, dependencies, and libraries.

Table 1: Workloads used in this paper (adapted from FunctionBench [28]). ***Note: The image size column denotes the size of the Docker image.***

Benchmark	Description	Image Size	Pull Latency (sec)
chameleon	Create an HTML table for a given number of columns and rows.	159 MB	2.03
json_serdes	Perform *json* deserialization of a given *json* file	160 MB	2.17
gzip	Perform gzip compression on a given file	160 MB	2.38
pyaes	Perform encryption and decryption using a private key-based mechanism.	163 MB	2.49
matmul	Perform matrix multiplication of two square matrices.	232 MB	4.77
image_rotate	Perform a rotation of a given image.	248 MB	6.26
lr_training	Train a model for a given dataset and target variable using logistic regression model.	618 MB	18.42
face_detection	Perform face detection on a given image using the Harcascade classifier.	630 MB	21.26
rnn	Generate a word using a RNN model.	995 MB	32.07
cnn	Classify an image using the Squeezenet model.	1.24 GB	43.85

Table 2: System configuration

Hardware settings			
Processor	Intel Xeon 6226R CPU, 2.90 GHz		
CPUs	1 Socket, 16 cores	DRAM	256 GB
HDD	Seagate 2TB SATA-3 HDD, 7200RPM		
SSD	Samsung 860 EVO 500GB SATA-3 SSD		
Software settings			
Linux kernel	v6.5-rc7	Go version	v1.20.13
Python	v3.10	Docker version	v23.0.3

3.1 Structure of a Function Image

Since serverless platforms execute hundreds of functions on a single node, the image pull latency can delay the response latency [1, 43, 59]. The image (*root* filesystem) of a function typically contains the following entities: operating system-related files, the runtime environment, function-specific dependencies (packages and libraries), and the source code. First, we formulate a standard layer organization of function images that can improve image pull latency and overall storage utilization.

According to a recent survey on AWS Lambda, about 44% of AWS Lambda invocations use the Node.js runtime, while 28% use the Python runtime [14]. AWS Lambda presently supports 3 versions of Node.js (v16, v18, and v20) and 5 versions of Python (v3.8, v3.9, v3.10, v3.11, and v3.12) [4]. With this insight, we divide a function image into two parts to leverage the overlap between layers of function images: a base segment and an app segment (see Figure 3). The base segment consists of common system and runtime-related files, while the *app* segment includes runtime libraries, function-specific dependencies, and source code. This partition ensures better overlapping between layers of function images. *The function images in this paper use the above-described image structure, which includes the following components: Python v3.10, Pip v23.0.1, and Debian 12.*

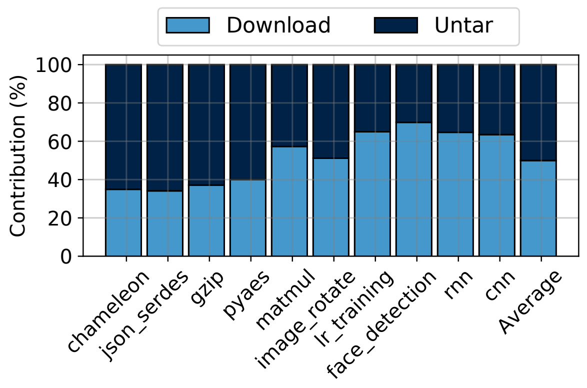


Figure 4: The relative contribution of layer downloading and layer untarring operations during function image pull.

3.2 Performance Analysis of an Image

When a platform receives a request to execute a serverless function, it pulls the function image from a remote registry. Prior work has shown that the image pull latency affects the response latency of the function [43, 56, 59]. In Table 1, we report the image pull latency of function images. To ensure fair comparison across images, we ensure that the base segment is available on a machine prior to the pull operation. Prior work has shown that downloading compressed layers and running the untar operation on the compressed layers contribute almost 80% to the container deployment latency [53]. Let us study the contribution of the aforementioned operations to the sum of both (referred to as the *relative contribution* in this paper).

In Figure 4, we show the relative contribution of layer downloading and layer untarring operations. We observe that layer downloading and layer untarring operations relatively contribute about 51% and 49% on average during image pull, respectively. We observe that the layer untarring operation is the bottleneck for smaller images. On the other hand, the layer downloading operation is the bottleneck for larger images. As a result, an image pull pipeline must focus on optimizing both download and untar operations.

As an image's size increases, the cost of layer downloading operations dominates that of layer untarring operations.

3.3 Layer Fusion vs On-Demand Download

To improve the pull latency of a function image, prior work proposed two techniques: layer fusion [19, 34], and on-demand download techniques [18, 23]. The former downloads all the files of an image, whereas the latter downloads files on demand. Now, let us answer the following research question in the context of serverless platforms: *"Which option is better for a function instance: including all the files in the image or fetching the files on-demand?"*.

3.3.1 File Analysis of a Function Image. Figure 5 shows that a container accesses only a portion of an image during the startup phase. Its size ranges from 12.5% to 64.7% of the image's size for the functions listed in Table 1. This is because serverless platforms operate at the granularity of a function, which does not utilize all of the features provided by libraries. For machine learning (ML) functions such as lr_training, rnn and cnn, a container accesses at least 37.75% of an image. In contrast, for non-ML functions, a container accesses at most 33.75% of an image. We also observed that during the execution phase, a container accesses up to 22 more files than those accessed during the startup phase. This is because during startup, a container imports the packages necessary for the execution of a function. We can conclude that a container accesses the majority of files during the startup phase. Therefore, the option of including all the files in an image is not better for all functions.

Furthermore, we note that containers of a function image consistently access the same set of files during the startup phase. This is because every container executes the same startup command as listed in the function image. The startup command accesses the required files to start the runtime, library files, the code, and function dependencies.

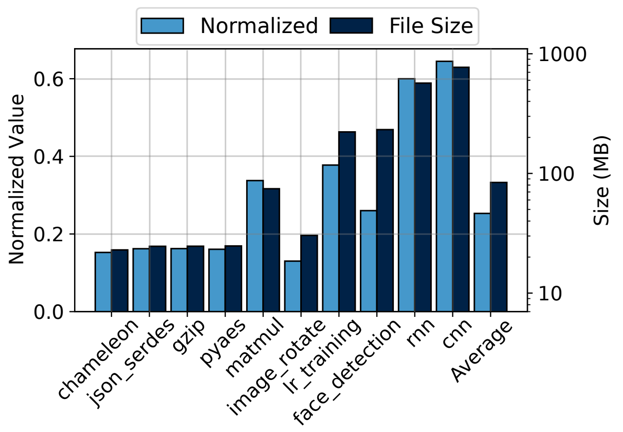


Figure 5: The size of the portion of an image accessed during startup is normalized to the image size. ***Note: A container accesses approximately 25.3% of the image size during startup.***

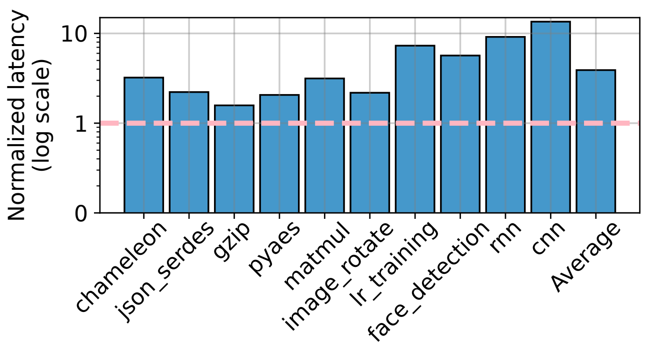


Figure 6: The startup latency of a function when using OpenFaaS + Gear normalized to that of OpenFaaS + Docker. ***The startup latency increases by at most 13.5× due to on-demand download of files.***

3.3.2 Efficacy of On-Demand Download Schemes. To evaluate the efficacy of on-demand download techniques, we integrate OpenFaaS with Gear [18] and analyze the startup latency of a function, given that images are deployed in increasing order of their size in the system (similar to SnapStore [41]). In Figure 6, we show that the function's startup latency increases by 3.9× compared to OpenFaaS + Docker on average. This is because downloading the files from a remote server on-demand degrades its latency. In the case of machine learning (ML) functions such as lr_training, rnn and cnn, a container accesses up to 38.7% of the total files, representing 64.7% of an image. In contrast, for basic functions like chameleon, json_serdes, gzip, and pyaes, a container accesses up to 12.5% of the total files, equating to 16.8% of the image size. We can conclude that fetching an image's files on-demand is also suboptimal.

(1) With layer fusion, all files are included in a single image, but a container only accesses about 12.5% to 64.7% of the image size for the functions listed in Table 1. This leads to resource wastage during the image pull process.
(2) When we utilize on-demand download, we retrieve a file whenever a container accesses it. However, a container might access 64.7% of the image size, leading to higher startup latency.
(3) Containers of a function image access the same set of files during the startup phase.

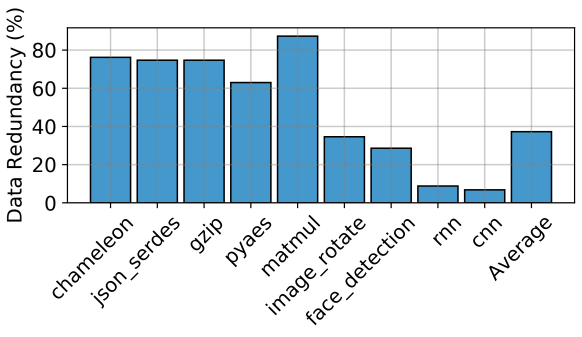


Figure 7: Data redundancy between the libraries of function images when compared with lr_training using the file-level deduplication scheme.

3.4 Redundancy Analysis of an Image

To improve the image pull operation, prior work focused on maximizing the overlapping of layers across functions. This will reduce the cost of the image pull operation by pulling the common layers only once. In this section, we study the source of redundancy between the function images. To find redundancy, we employ the file-level deduplication scheme (similar to Break [19] and Gear [18]). In this scheme, we identify identical files by comparing the hash value of the file's contents across different files. The redundancy of func image $\mathcal{B}$ with respect to func image $\mathcal{A}$ is the percentage of redundant data available in $\mathcal{B}$ [41].

As per our proposed layer organization of function images, the base segment remains the same for all the function images having the same runtime environment. So, we focus on the app segment of a function image. Intuitively, we further split the app segment into two layers: *func* (dependencies + code) and *lib* (runtime libraries). We then compare the redundancy of both layers of function images in comparison to that of the lr_training (representative) function. In Figure 7, we observe that the average pair-wise redundancy of the *lib* layer is 37.3%. This is because multiple functions use a common set of libraries, such as Flask, Werkzeug, and Jinga. However, the *func* layer has no redundancy because the functions do not share any common functionality.

For the functions listed in Table 1, the average pair-wise redundancy between the libraries of function images is 37.3%.

4 Design

In this section, we discuss the design of FaaSImage, a novel function image manager that manages the build, pull, and mount operations of a function image. Furthermore, we explain that FaaSImage can be seamlessly integrated with serverless platforms and is compatible with the existing container I/O stack.

4.1 Design Principles

In Section 3, we have shown that it is not necessary to store all the files in an image, as only a fraction of the image is utilized during the startup and execution phases. Furthermore, we demonstrated that the files of a function image can be divided into three layers based on redundancy across function images: *base*, *func* and *lib*. The base layer stores common operating system and runtime environment-related files across images. The func layer stores the code and dependencies that are unique to functions. The lib layer stores the shared libraries across function images. These lead to the following design decisions:

- An image should store the necessary files for the startup phase and fetch the remaining files on demand during the execution phase.
- To leverage the redundancy between libraries of function images, we create a *lib layer* that is shared across function images, thereby decoupling libraries from an image.
- After decoupling, FaaSImage splits the files of an image into two layers: *base* and *func*.
- FaaSImage can fetch the func layer and necessary library files missing in the lib layer in parallel. Note that the base layer is common across function images and fetched only once (need not be optimized a lot).

4.2 Overview

In Figure 8, we show the high-level architecture of FaaSImage. FaaSImage consists of four components: an image builder, an image registry, a storage manager, and the FAAS overlay module. The *image builder* is responsible for creating a function image and finding the required files for the startup phase of a function container. The *image registry* is a central server that decouples libraries from an image, stores the libraries on the server, and refactors the processed image into two layers: *base* and *func*. During an image pull operation from the registry, the *storage manager* retrieves the func

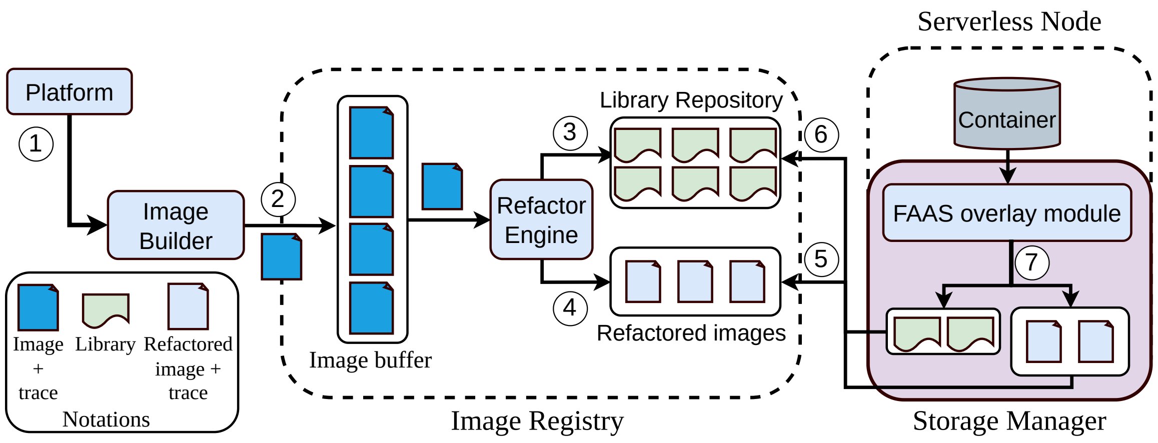


Figure 8: The high-level architecture of FaaSImage.

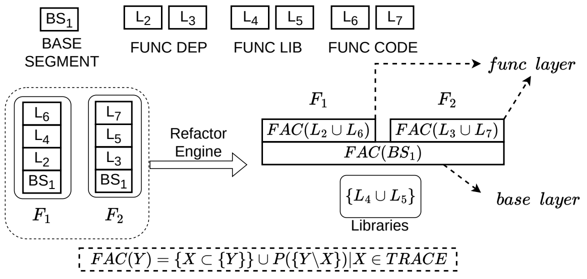


Figure 9: The illustration of refactoring a given set of images into a common *base* layer, unique *func* layers, and shared libraries by FaaSImage.

layer and necessary library files missing in the *lib* layer in parallel. During container deployment, the *FAAS overlay module* creates a mount point for a container, providing a merged view of the layers and libraries. Additionally, it contains an on-demand download manager, which downloads the missing files from a remote server during the execution phase. Let us now discuss the build, pull, and mount operations of FaaSImage at a high level.

When a user registers a function, the platform builds the function image with the aforementioned structure (as discussed in Section 3.1) (see ① in Figure 8). Subsequently, we start a function instance on a node, which is ready to accept function requests. Simultaneously, we initiate the file access tracker, which logs the files that are accessed during startup (referred to as *trace* in this paper). Finally, we send the image and the trace files to a registry (see ② in Figure 8).

The registry's refactor engine mounts the function image into a directory. Subsequently, it extracts the third-party libraries utilized by a function from the image and stores them in the library repository (see ③ in Figure 8). For the remaining files, we selectively retain only the required files needed for startup as per the trace file and replace the contents of the unnecessary files with their MD5 hash. In the future, the FAAS overlay module may use a file's MD5 hash to retrieve its contents on demand during the execution phase. Then we split the files into two layers, *base* and *func*, and create a refactored image (see ④ in Figure 8). This step concludes the re-packaging of the function image in the registry.

While serving a function request, the serverless node retrieves the function's image from the registry. The storage manager first retrieves the `manifest.json`, as well as the trace file. Then it analyzes the trace file to find the required files for the respective libraries. Subsequently, it sends a query to the library repository to fetch those files. Simultaneously, the layer manager fetches the base and func layers of the image (see ⑤ and ⑥ in Figure 8). Note that we can fetch the base layer only once in a system because it is the same across multiple functions. We mark the image download operation as successful once we have downloaded the libraries and layers. When a platform deploys a container using the function image, the FAAS overlay module provides a mount point to a container to access files (see ⑦ in Figure 8).

4.3 Image Builder

To register a function on a serverless platform, the user must provide the following details: memory configuration, the function's code, any function dependencies, and the desired runtime environment. The memory configuration specifies the amount of memory required by the function. The image builder builds the function image using the remaining details (as discussed in Section 3.1). It has been observed that a container does not utilize all of the files during the startup and execution phases (as analyzed in Section 3.3). Consequently, the notion of bundling all the files into an image is suboptimal. Let us discuss the file access tracker that identifies the files accessed by a container during the startup phase.

Prior to executing any file operation in a container, the kernel executes a lookup operation, which finds the inode number and the file descriptor of the file. This is because the operating system needs to find the file's exact location and ascertain that operations adhere to the file's access permissions. As a result, we extend the existing overlay kernel module by including a logger. The logger records every file lookup as a tuple $\langle abs\_path, PID \rangle$, where *abs_path* is the absolute file path and *PID* is the container process's PID accessing that file. Subsequently, the logger sends tuples to the file access tracker running in the userspace through a netlink socket. After

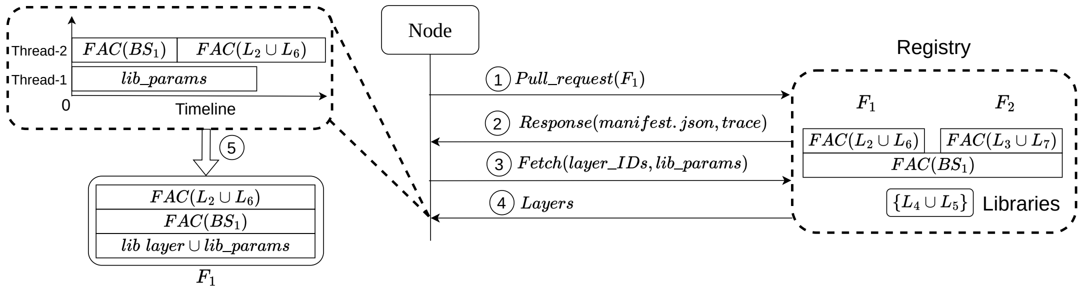


Figure 10: The process of pulling an image in FaaSImage.

Algorithm 1 Refactor an image

```
1:  procedure FAC($files$, $trace$, $layer$)
2:      for all file $f \in files$ do
3:          if $f \in trace$ then
4:              ADD_FILE($layer$, $f$)        ▷ Add file to the layer
5:          else
6:              ADD_FILE_HASH($layer$, $f$) ▷ Add the hash of a file's contents to the layer
7:          end if
8:      end for
9:  end procedure
10:
11: procedure REFACTOR_IMAGE($image\_id$, $trace$)
12:     $dir$ ← mount($image\_id$) ▷ Mount an image at directory dir
13:     $lib\_path$ ← /usr/local/lib/python3.10/site-packages/
14:     $lib\_set$, $base\_set$, $func\_set$ ← {}, {}, {}
15:     for all file $f \in dir$ do
16:         if $f \in base\ segment$ then
17:             INSERT($base\_set$, $f$)
18:         end if
19:         if $lib\_path \in$ FILE_PATH($f$) then
20:             $lib\_name$, $lib\_version$ ← GET_DETAILS($f$) ▷ Get the name and version of a library.
21:             ADD_FILE($lib\_name : lib\_version$, $f$)
22:             INSERT($lib\_set$, $lib\_name : lib\_version$)
23:         end if
24:         if $f \in app\ segment$ then
25:             INSERT($func\_set$, $f$)
26:         end if
27:     end for
28:     FAC($base\_set$, $trace$, $base\ layer$)
29:     FAC($func\_set$, $trace$, $func\ layer$)
30:     for all library $lib \in lib\_set$ do
31:         ADD_LIB_SYMLINK($func\ layer$, $lib$)        ▷ Create a library's symlink
32:     end for
33: end procedure
```

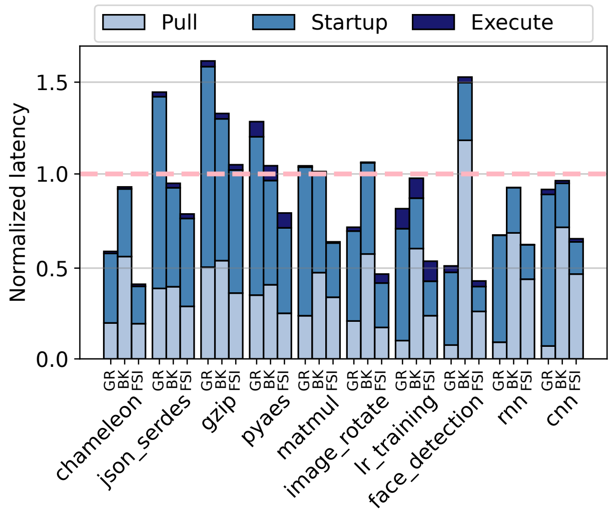
receiving the tuple, the file access tracker stores the *abs_path* in the *trace* file that corresponds to the container's image. *Note that the trace file represents the files needed for startup, not the execution, and is thereby not affected by input parameters.*

4.4 Image Registry

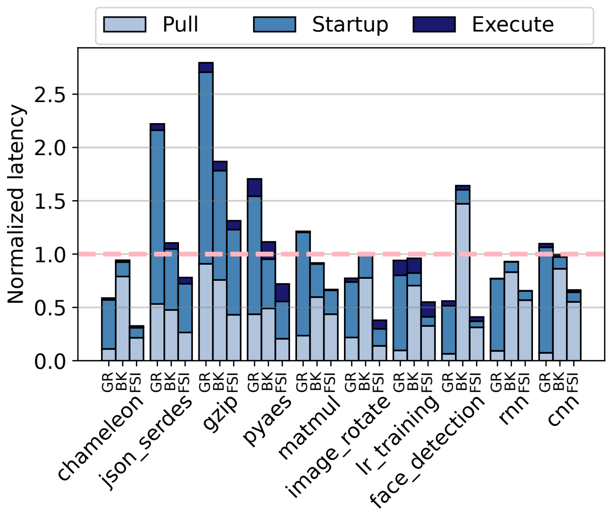
After building an image, the platform proceeds to upload both the image and the trace file to a centralized image registry. An image can be divided into two parts: the base segment and the app segment (discussed in Section 3.1). The base segment is common across multiple images, whereas the app segment contains the code, libraries and dependencies. We observe that a container accesses only a subset of the image's files during the startup phase, and the size of this subset depends on the characteristics of a function. Furthermore, we observe that the redundancy among multiple app segments comes from third-party libraries. With these insights, let us refactor an image so that the amount of data downloaded and extracted during an image pull operation decreases. *Note that if a node attempts to download an image during the refactor process, the registry sends a version of the function image stored in the image buffer.*

In Algorithm 1, we show the workings of the refactoring engine. We first define the *FAC* module, which iterates through all the files of a layer, selectively writing only those listed in the trace file (Lines-3-4). It replaces the contents of the remaining files with their content's MD5 hash that will be downloaded during execution (Lines-5-7). After receiving an image, the refactor engine first separates third-party libraries from the image and stores them in the "libraries partition" of the registry. This is because we plan to store only a single copy of a library that is shared across multiple images. We store the files of each library in a directory labeled as `<pkg_name>:<vX>` (Lines-19-23). As we separate the libraries from an image, we must keep some metadata in the image to identify the libraries that are a part of it. For each library `P` in the image, we create a symbolic link that connects the original path of library `P` to the directory `/P:<vX>/` in the image (see Lines-30-32). We append the list of library names and their versions associated with the image in the manifest.json file. Next, we refactor the base segment, function code and dependencies.

In Figure 9, we show a graphical illustration of the layers present in the image before and after the refactoring process. To minimize the number of download and untar operations, we merge the layers of the base segment into a single layer. Similarly, we merge the code and dependencies into a single layer. Subsequently, we process them through the *FAC* module, thereby generating the base layer and the func layer (see Lines-28-29). Finally, using both layers, we create the refactored function image that adheres to the open container initiative (OCI) standards. Note that the libraries



(a) Cold start latency on HDDs.



(b) Cold start latency on SSDs.

Figure 11: The cold start latency of Gear (GR), Break (BK), and FaaSImage (FSI) when function images are pulled in *Config-1* (normalized to that of OpenFaaS + Docker). *Takeaway: FaaSImage improves the cold start latency by 38.7% on HDDs and 39.8% on SSDs on an average.*

store trace-independent files because it is possible for two functions to access the same library.

4.5 Storage Manager

When the platform receives a function request, it pulls the function's image from the image registry, and then deploys a container to execute the function on a node. Figure 10 shows the overall image pull operation. During the pull operation, the node first fetches the following files: `manifest.json` and `trace`. The manifest.json file stores the list of layer IDs and the list of library names used by the function (see ②). Subsequently, it finds the files of libraries that are required for deployment but not present on the machine and generates a list of tuples $\langle library\_name, \{files\}\rangle$ (referred to as *lib_params*). Finally, the node sends a request to fetch layers from the registry with the following parameters: the list of layer IDs and *lib_params* (see ③).

Based on our refactoring mechanism, we know that the base layer is common across function images and fetched once, the func layer is unique across function images and libraries are shared across function images. FaaSImage extends the pull pipeline of Docker with an additional thread that fetches the *lib_params*. Consequently, Docker pulls the function layer and *lib_params* in parallel, thereby reducing the image pull latency (see ④). The libraries' files are stored in the /var/lib/docker/libraries directory (referred to as the *lib* layer in this paper) (see ⑤). This leads to faster image pull latency and lower bandwidth utilization.

4.6 FAAS Overlay Module

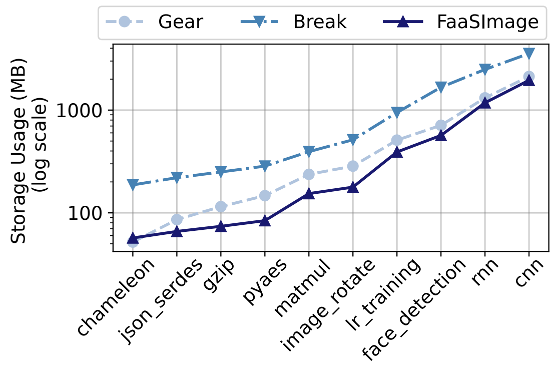
After the image pull operation is completed, Docker creates a container mount point using the FAAS overlay kernel module so that a container can access files. The FAAS overlay module mounts the layers of an image in the following order: *lib*, *base*, and *func*. The FAAS overlay module is based on the overlay2 storage driver, thereby compatible with the current Docker I/O stack. As we are only storing the required files for the startup phase, a container might access additional files during the execution phase that are not part of an image. We identify those files during the file lookup operation (`ovl_lookup()`). Then, the overlay kernel module communicates the file's MD5 hash and the file path to a userspace application. Subsequently, the userspace application fetches the file from a remote server and overwrites the missing file's contents.

5 Evaluation

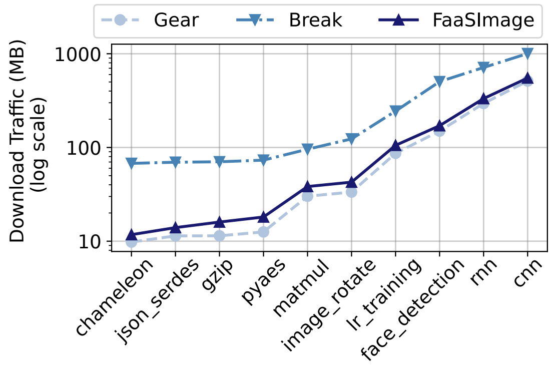
In this section, we discuss the efficacy of FaaSImage for storing and retrieving function images on a serverless node. We evaluate FaaSImage by comparing its cold start latency, pull latency, download traffic and storage cost integrated with OpenFaaS against two state-of-the-art schemes: Gear [18] and Break [19].

5.1 Evaluation setup

Our serverless node setup has been described in Table 2 (see Section 3). To host the image registry, we use a machine with an Intel Core i5-4590 CPU, 4 CPU cores, 1 TB Seagate hard disk, and 12 GB of memory. The average ping latency between the node and the registry is 445 μsec. Prior work has evaluated their schemes on popular Docker images or Docker image datasets [18, 19, 41]. We use the workloads from FunctionBench described in Table 1 (implemented using the Python runtime environment). Moreover, on the lines of prior work, we select the images in increasing order of their image size as our dataset (referred to as *Config-1* in the paper) [41]. This ensures that we have a standard baseline. In addition, we evaluate our scheme on the following three runtime environments: Node.js v18 [4], PHP v8.3 [22] and Ruby v3.3 [4] to show that our scheme is also effective for other runtime environments. Note that we store the auxiliary files such as library packages and other files needed by the different schemes in the registry node.



(a) Storage utilization of function images.



(b) Download traffic while serving a function request.

Figure 12: The storage usage and download traffic experienced by a serverless node, when we execute functions in *Config-1*.

We have seamlessly integrated FaaSImage into Docker v23.0.3 and Linux Kernel 6.5-rc7 by using about 2200 lines of Go code and 350 lines of C code. With respect to Docker, the image refactoring engine and the file server of FaaSImage are standalone components. We have extended the `pullSchema2-Layers()` function in *pull_v2.go* of Docker with a Go routine that downloads the *lib_params* files. Furthermore, we have made modifications to the `ovl_lookup()` function in *namei.c* of the overlay kernel module to download missing files.

5.2 Cold Start Performance

In this section, we evaluate the cold start latency of FaaSImage when integrated with OpenFaaS. When a platform serves a function request, the cold start latency captures the following steps: ① pull an image from a remote registry, ② start a container using the image, and ③ execute a function request. We compare the cold start latency of FaaSImage, Gear, and Break. In addition to measuring the cold start latency, we also measure the cumulative download traffic and storage footprint of function images. For fair comparison, we ensure that a node does not contain any auxiliary files or library's files prior to the pull operation.

5.2.1 Overall Performance. In Figure 11, we show that FaaSImage improves the cold start latency by 38.7% on HDDs and 39.8% on SSDs when compared against OpenFaaS + Docker on average, respectively. This is because FaaSImage improves the pull latency and ensures that a minimal number of files are downloaded during a function's execution. In the case of Gear, the startup latency accounts for at least 74.01% of the cold start latency. This is because Gear fetches the files of an image when it is accessed by a container. For the *face_detection* workload, we observe that Break performs worst compared to other works. This is because Break refactors the workload's image such that one layer contains about 77% of the image, thereby causing stalls in the pull pipeline.

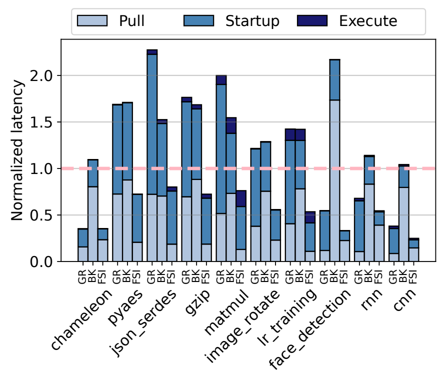
5.2.2 Pull Performance. The pull operation consists primarily of two steps: downloading compressed layers and untarring compressed layers. FaaSImage improves the pull latency by 52% on HDDs and 58% on SSDs when compared against Break (in Config-1). This is because FaaSImage undertakes a hybrid approach where we only fetch a portion of the image, and the remaining portion is fetched during function execution (unlike Break and Gear). As a result, the download latency and untar latency decrease at least by an average of 13% and 62%, respectively.

On the other hand, we observe that the pull latency of FaaSImage increases by 68% on HDDs and 1.05× on SSDs when compared against Gear on average. Gear has the lowest image pull latency because it uses the on-demand download technique; it only pulls file pointers to a remote storage machine. As a result, the download latency of FaaSImage increases by at most 54% when compared against Gear. For *gzip*, FaaSImage outperforms Gear by 52.5% in terms of the pull latency, as Gear's single-layer structure stores all file pointers within one layer. Consequently, the cost of the layer untar operation is 4× higher than that of FaaSImage.

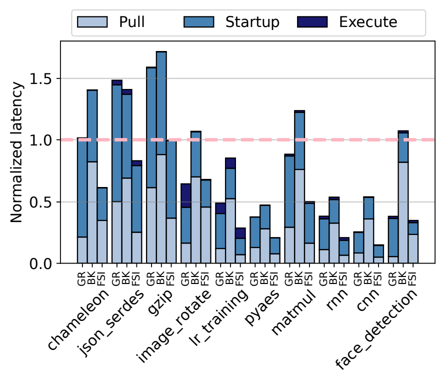
To study the effect of the ordering of function images on the pull latency, we randomly generate 40 sequences of function images (enough to reach a steady state). We observe that the image pull latency improves by at most 57.5%. The standard deviation of the improvement across all storage technologies is in the ballpark of 1.7%. Furthermore, we performed a limit study that concurrently downloads multiple images from the registry; we see that the pull latency increases by 3.1× for *face_detection* (representative workload) when 10 images are pulled together.

5.2.3 Execution Analysis. In the case of FaaSImage, we observe that a container downloads 18 files during the execution of the *image_rotate* benchmark and 4 files during the execution of the *cnn* benchmark. As a result, the execution latency of *image_rotate* increases from 0.04s to 0.45s, while that of *cnn* increases from 0.76s to 0.91s when compared against OpenFaaS + Docker on HDDs. For *image_rotate*, FaaSImage downloads the PIL package during the function execution and saves the files with the current timestamp. As a result, Python rebuilds __pycache__ for the package, increasing the execution time. In contrast, Gear saves the files with the same timestamp from the image build, allowing it to reuse the existing __pycache__, which speeds up execution.

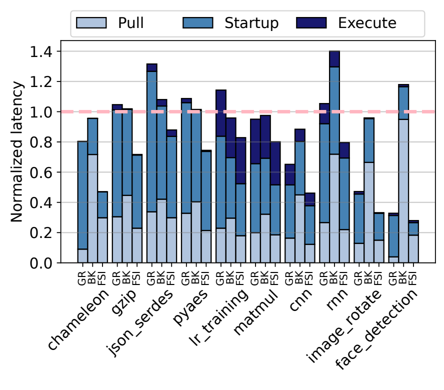
5.2.4 Storage and Download Traffic. In Figure 12, we show that FaaSImage and Gear improve the storage usage and download traffic by at least 40% and 45% when compared to Break, respectively.



(a) Node.js runtime.

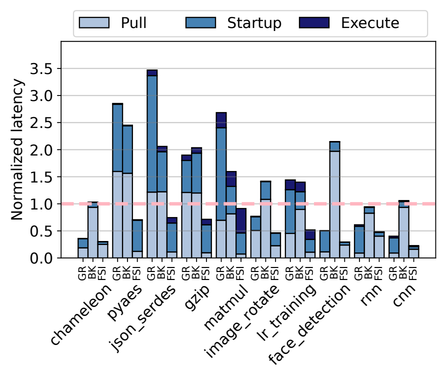


(b) Ruby runtime.

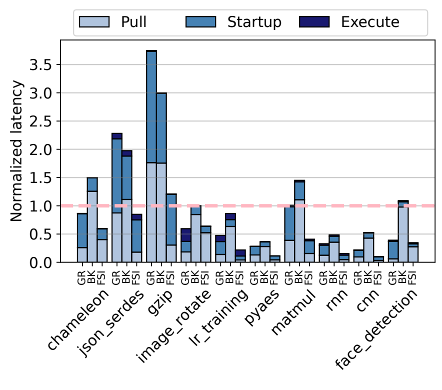


(c) PHP runtime.

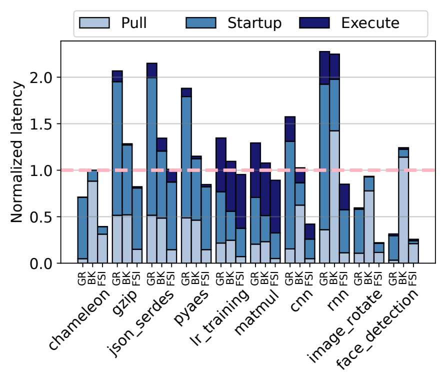
Figure 13: The cold start latency of Gear (GR), Break (BK), and FaaSImage (FSI) when function images of different runtime environments are pulled in *Config-1* on HDDs (normalized to that of OpenFaaS + Docker). *Takeaway: FaaSImage improves the cold start latency for Node.js, Ruby and PHP runtimes by 49.9%, 38.9% and 31.1% on an average, respectively.*



(a) Node.js runtime.



(b) Ruby runtime.



(c) PHP runtime.

Figure 14: The cold start latency of Gear (GR), Break (BK), and FaaSImage (FSI) when function images of different runtime environments are pulled in *Config-1* on SSDs (normalized to that of OpenFaaS + Docker). *Takeaway: FaaSImage improves the cold start latency for Node.js, Ruby and PHP runtimes by 50.9%, 49.1% and 42.5% on an average, respectively.*

This is because FaaSImage and Gear only store the required files necessary for function deployment and execution. On the other hand, FaaSImage improves storage usage by 8% when compared against Gear. This is because FaaSImage shares the base layer across function images, while Gear creates a single layer for every function image. However, FaaSImage increases the download traffic by 8% against Gear. This is because Gear uses file-level deduplication across image files, thereby improving download traffic. Furthermore, we observe that FaaSImage increases the CPU utilization of the Docker process by 1.22%.

5.3 Performance Across Different Runtimes

In this section, we show that our scheme is effective for other runtime environments. For this analysis, we port the serverless functions in Table 1 to three additional runtime environments: Node.js v18, PHP v8.3 and Ruby v3.3. In Figures 13 and 14, we show that FaaSImage improves the average cold start latency of functions implemented in other runtime environments by at least 31.1% on HDDs and 42.5% on SSDs, respectively (in Config-1). This is because FaaSImage improves the pull latency by fetching the necessary files and retrieving the missing files during execution.

5.3.1 *Pull Performance.*

FaaSImage improves the pull latency by at least 60.2% on HDDs and 80.1% on SSDs (resp.) as compared to Break. This is because Break splits most of these images into two layers: a common layer (pulled once for the runtime environment) and a unique layer. During image pull operations, Break downloads the unique layer and then untars it. In contrast, FaaSImage selectively downloads only the files essential for the startup phase, reducing layer download and untar operations by 74.6% and 60.7% on HDDs, and by 85.1% and 71.2% on SSDs, respectively. Furthermore, we also observe that Gear's pull latency is higher than FaaSImage for certain function images due to the single-layer structure of Gear, which stores all file pointers within one layer. In the case of Node.js images, which contain between 37.2K and 53.8K files, FaaSImage reduces the average cost of untar operations by 74.4% compared to Gear.

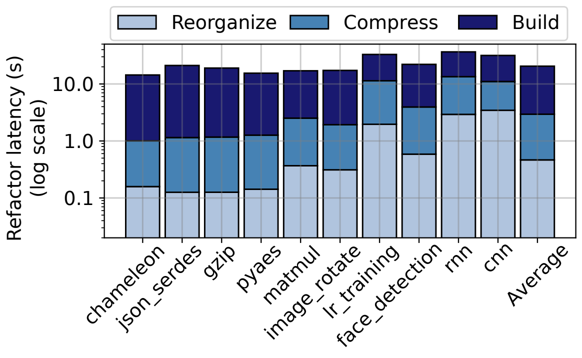


Figure 15: The time taken to convert a function image into the FaaSImage format on HDDs.

5.4 Image Refactoring Time

Prior to pulling an image, we need to perform the refactor operation on the image to convert it into the FaaSImage format. The refactor operation can be divided into three steps: ① subsetting and reorganizing files into different layers, ② compressing a layer directory, and ③ building the corresponding Docker image. In Figure 15, we show the time taken to refactor function images into the corresponding FaaSImage format. We observe that the image refactor latency is 21.6 seconds, on average. This is because the Docker build operation contributes about 81.4% to the refactoring latency. Furthermore, we observe that the refactor latency is proportional to the total size of files that a function's container accesses during the startup phase. Overall, the refactor latency is acceptable as the refactor operation is performed only once for function images (similar to Gear [18]), and it is off the critical path.

6 Related Work

As microservice and serverless paradigms are gaining popularity, developers are increasingly deploying their applications on lightweight sandboxes such as Docker [6, 25, 38, 39, 51, 55]. Prior work has shown that the cost of fetching the container's image is the primary factor affecting cold start [43, 56, 57, 59]. Notably, prior work has proposed peer-to-peer mechanisms [17, 56, 60], container/sandbox caching [20, 40, 50] and pre-provisioning mechanisms [37, 42, 61] for serverless platforms to minimize the deployment overhead [39, 57]. Our method is complementary, as we focus on image engineering to reduce the cold start latency by restructuring images [18, 19, 56].

6.1 Layer Fusion

When Docker pulls an image, it only fetches the layers that are unavailable on the host machine, thereby improving the image pull operation. However, there is significant redundancy between the unique layers of different images, which leads to redundant files being downloaded multiple times [49]. Li et al. [34] proposed an image reconstruction algorithm that increases the number of identical layers across images, but they did not consider the overhead of the untarring operation. In contrast, Break [19] applied the idea of Venn diagram based construction to refactor images in a way that stores shared files in a single layer. Furthermore, it proposed that the untarring operation can be done in parallel across multiple layers of an image to further reduce the latency of the pull operation. However, if we encounter layers with skewed sizes, parallelizing the untar operation will not be effective.

In the case of serverless workloads, a function container does not access all the files in an image. As a result, the idea of packaging all the files in an image leads to a wastage of resources. FaaSImage uses subsetting to retain the necessary files of an image required for the startup phase and share libraries between function images, thereby reducing the size of an image and the pull latency.

6.2 On-Demand Download

According to Slacker [23], a container utilizes a small fraction of the image at the time of deployment. This suggests that pulling the complete image is not efficient. To address this issue, Slacker uses the checkpoint and clone operations provided by Tintri VMstore [54] to mount the image on a machine and fetch files on demand. In contrast, Nyudus [45] uses a user-level filesystem to pull files at the granularity of chunks, while DADI [33] uses a virtual block device to pull files at the block level. However, these mechanisms require significant changes to the existing container I/O stack, which limits their adaptability. Gear [18] converts an existing image into a single layer and replaces the contents of each file with a pointer to the file on the remote server. When a container accesses a file, it sends a request to the remote server if the file is unavailable on the host machine. This design is compatible with the existing container I/O stack. In the case of serverless functions, the size of a fraction utilized by a container is function dependent; it ranges from 12.5-64.7% of the image size. As a result, the cost of fetching files of an image increases the startup latency of functions. FaaSImage relies on on-demand downloading of files during the function execution and only stores the necessary files required for the startup phase.

7 Conclusion

This paper proposes FaaSImage, a novel image manager designed specifically for the serverless computing paradigm. Our comprehensive approach integrates layer fusion, on-demand download, and layer subsetting techniques to improve the cold start latency of serverless functions. FaaSImage improves the image pull latency by at most 52% on HDDs and 58% on SSDs over the state-of-the-art approaches. Upon integration with OpenFaaS, FaaSImage improves the cold start latency by 38.7% on HDDs and 39.8% on SSDs when compared against OpenFaaS. Furthermore, FaaSImage reduces the storage space, and download traffic by at most 45% when compared against state-of-the-art approaches.

Acknowledgments

This work was supported by the Prime Minister's Research Fellows (PMRF) scheme, and partially supported by the CSE Research Acceleration Fund of IIT Delhi.

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: lightweight virtualization for serverless applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 419–434.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 923–935.

[3] Amazon. 2024. AWS Lambda – Case Studies. (2024). https://aws.amazon.com/lambda/resources/customer-case-studies/

[4] Amazon. 2024. Lambda runtimes - AWS Lambda. (2024). https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html

[5] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS Made Fast Using Snapshot-Based VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 730–746. https://doi.org/10.1145/3492321.3524270

[6] Apache. 2024. Apache OpenWhisk is a serverless, open source cloud platform. (2024). https://openwhisk.apache.org/

[7] Microsoft Azure. 2023. *Azure Functions – Serverless Functions in Computing*. Retrieved May 25, 2023 from https://azure.microsoft.com/en-us/products/functions

[8] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand Container Loading in AWS Lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/atc23/presentation/brooker

[9] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698

[10] Google Cloud. 2023. *Cloud Functions*. Retrieved May 25, 2023 from https://cloud.google.com/functions

[11] IBM Cloud. 2023. *IBM Cloud Functions*. Retrieved May 25, 2023 from https://cloud.ibm.com/functions

[12] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 64–78. https://doi.org/10.1145/3464298.3476133

[13] Datadog. 2024. *The Docker instance should not use AUFS as its storage driver*. Retrieved May 1, 2024 from https://docs.datadoghq.com/security/default_rules/cis-docker-1.2.0-2.5/

[14] Datahog. 2023. The state of serverless. (2023). https://www.datadoghq.com/state-of-serverless/

[15] Docker. 2024. *docker pull | Docker Docs*. Retrieved May 1, 2024 from https://docs.docker.com/reference/cli/docker/image/pull/

[16] Docker. 2024. *Docker storage drivers | Docker Docs*. Retrieved May 1, 2024 from https://docs.docker.com/storage/storagedriver/select-storage-driver/

[17] Dragonfly. 2024. *Introduction | Dragonfly*. Retrieved May 1, 2024 from https://d7y.io/docs/

[18] Hao Fan, Shengwei Bian, Song Wu, Song Jiang, Shadi Ibrahim, and Hai Jin. 2021. Gear: Enable Efficient Container Storage and Deployment with a New Image Format. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 115–125. https://doi.org/10.1109/ICDCS51616.2021.00020

[19] Yicheng Feng, Shihao Shen, Xiaofei Wang, Qiao Xiang, Hong Xu, Chenren Xu, and Wenyu Wang. 2024. BREAK: A Holistic Approach for Efficient Container Deployment among Edge Clouds. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*. 1491–1500. https://doi.org/10.1109/INFOCOM52122.2024.10621084

[20] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 386–400. https://doi.org/10.1145/3445814.3446757

[21] Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. 2017. Shifter: Containers for HPC. *Journal of Physics: Conference Series* 898, 8, 082021. https://doi.org/10.1088/1742-6596/898/8/082021

[22] Google. 2024. Runtime Support | Cloud Run functions Documentation | Google Cloud. (2024). https://cloud.google.com/functions/docs/runtime-support

[23] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: fast distribution with lazy docker containers. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'16)*. USENIX Association, USA, 181–195.

[24] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless computation with openLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (Denver, CO) *(HotCloud'16)*. USENIX Association, USA, 33–39.

[25] David Jaramillo, Duy V Nguyen, and Robert Smart. 2016. Leveraging microservices architecture by using Docker technology. In *SoutheastCon 2016*. 1–5. https://doi.org/10.1109/SECON.2016.7506647

[26] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[28] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 502–504. https://doi.org/10.1109/CLOUD.2019.00091

[29] Knative. 2024. *Mitigate Eliminate image pull latency*. Retrieved May 1, 2024 from https://github.com/knative/serving/issues/283/

[30] Kraken. 2024. *kraken: P2P Docker registry capable of distributing TBs of data in seconds*. Retrieved May 1, 2024 from https://github.com/uber/kraken/

[31] AWS Lambda. 2023. *Lambda execution environments - AWS Lambda*. Retrieved May 19, 2023 from https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html

[32] AWS Lambda. 2023. *Serverless Computing - Amazon Web Services*. Retrieved May 25, 2023 from https://aws.amazon.com/lambda/#:~:text=AWS%20Lambda%20is%20a%20serverless,pay%20for%20what%20you%20use

[33] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. DADI: block-level image service for agile and elastic application deployment. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 50, 14 pages.

[34] Sisi Li, Ao Zhou, Xiao Ma, Mengwei Xu, and Shangguang Wang. 2022. Commutativity-guaranteed docker image reconstruction towards effective layer sharing. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France). Association for Computing Machinery, New York, NY, USA, 3358–3366. https://doi.org/10.1145/3485447.3512154

[35] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. 2021. FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (Virtual Event, Germany) *(PLOS '21)*. Association for Computing Machinery, New York, NY, USA, 38–45. https://doi.org/10.1145/3477113.3487273

[36] Openfaas Ltd. 2024. *Home | OpenFaaS - Serverless Functions Made Simple*. Retrieved May 18, 2024 from https://www.openfaas.com/

[37] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[38] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing* (Renton, WA, USA) *(HotCloud'19)*. USENIX Association, USA, 21.

[39] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 57–69.

[40] Li Pan, Lin Wang, Shutong Chen, and Fangming Liu. 2022. Retention-Aware Container Caching for Serverless Edge Computing. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 1069–1078. https://doi.org/10.1109/INFOCOM48880.2022.9796705

[41] Abhisek Panda and Smruti R. Sarangi. 2023. SnapStore: A Snapshot Storage System for Serverless Systems. In *Proceedings of the 24th International Middleware Conference* (Bologna, Italy) *(Middleware '23)*. Association for Computing Machinery, New York, NY, USA, 261–274. https://doi.org/10.1145/3590140.3629120

[42] Manish Pandey and Young-Woo Kwon. 2024. FuncMem: Reducing Cold Start Latency in Serverless Computing Through Memory Prediction and Adaptive Task Execution. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing* (Avila, Spain) *(SAC '24)*. Association for Computing Machinery, New York, NY, USA, 131–138. https://doi.org/10.1145/3605098.3636033

[43] Shijun Qin, Heng Wu, Yuewen Wu, Bowen Yan, Yuanjia Xu, and Wenbo Zhang. 2020. Nuka: A Generic Engine with Millisecond Initialization for Serverless Computing. In *2020 IEEE International Conference on Joint Cloud Computing*. 78–85. https://doi.org/10.1109/JCC49151.2020.00021

[44] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for

Computing Machinery, New York, NY, USA, 753–767. https://doi.org/10.1145/3503222.3507750

[45] Nydus Image Service. 2024. *Nydus : the Dragonfly image service, providing fast, secure and easy access to container images.* Retrieved May 25, 2023 from https://github.com/dragonflyoss/nydus

[46] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 14, 14 pages.

[47] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. 2022. Fireworks: A Fast, Efficient, and Safe Serverless Framework Using VM-Level Post-JIT Snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 663–677. https://doi.org/10.1145/3492321.3519581

[48] Harsh Singh. 2024. *Docker does not extract images in parallel · Issue #21814 · mobymoby.* Retrieved May 1, 2024 from https://github.com/moby/moby/issues/21814

[49] Dimitris Skourtis, Lukas Rupprecht, Vasily Tarasov, and Nimrod Megiddo. 2019. Carving perfect layers out of Docker images. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing* (Renton, WA, USA) *(HotCloud'19)*. USENIX Association, USA, 17.

[50] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2024. UniCache: The Next 700 Caches for Serverless Computing. In *International Workshop on Cloud Intelligence/AIOps (AIOps '24)*.

[51] Joe Stubbs, Walter Moreira, and Rion Dooley. 2015. Distributed Systems of Microservices Using Docker and Serfnode. In *2015 7th International Workshop on Science Gateways*. 34–39. https://doi.org/10.1109/IWSG.2015.16

[52] Markus Thömmes. 2017. *Squeezing the milliseconds: How to make serverless platforms blazing fast!* Retrieved May 25, 2023 from https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0

[53] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[54] VMstore. 2024. *Tintri VMstore™ T7000 NVMe Platform.* Retrieved May 1, 2024 from https://tintri.com/wp-content/uploads/2023/01/data-sheet-009-vmstore-t7000.pdf

[55] Xili Wan, Xinjie Guan, Tianjing Wang, Guangwei Bai, and Baek-Yong Choi. 2018. Application deployment using Microservice and Docker containers: Framework and optimization. *Journal of Network and Computer Applications* 119 (2018), 97–109. https://doi.org/10.1016/j.jnca.2018.07.003

[56] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[57] Bowen Yan, Heran Gao, Heng Wu, Wenbo Zhang, Lei Hua, and Tao Huang. 2021. Hermes: Efficient Cache Management for Container-based Serverless Computing. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware* (Singapore, Singapore) *(Internetware '20)*. Association for Computing Machinery, New York, NY, USA, 136–145. https://doi.org/10.1145/3457913.3457925

[58] 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 *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 335–350. https://doi.org/10.1145/3617232.3624871

[59] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/3419111.3421280

[60] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. 2018. Wharf: Sharing Docker Images in a Distributed File System. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 174–185. https://doi.org/10.1145/3267809.3267836

[61] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3567955.3567960