BREAK: A Holistic Approach for Efficient Container Deployment among Edge Clouds

Yicheng Feng[†], Shihao Shen[†], Xiaofei Wang[†*], Qiao Xiang[‡], Hong Xu[§], Chenren Xu[◇], Wenyu Wang[○]
[†]Tianjin University, [‡]Xiamen University, [§]The Chinese University of Hong Kong, [◇]Peking University, [○]PPIO Cloud Computing (Shanghai) Co., Ltd.
{yichengfeng, shenshihao, xiaofeiwang}@tju.edu.cn, qiaoxiang@xmu.edu.cn, hongxu@cuhk.edu.hk, chenren.xu@gmail.com, wayne@pplabs.org

***Abstract*—Container technology has revolutionized service deployment, offering streamlined processes and enabling container orchestration platforms to manage a growing number of container clusters. However, the deployment of containers in distributed edge clusters presents challenges due to their unique characteristics, such as bandwidth limitations and resource constraints. Existing approaches designed for cloud environments often fall short in addressing the specific requirements of edge computing. Additionally, very few edge-oriented solutions explore fundamental changes to the container design, resulting in difficulties achieving backward compatibility.**

In this paper, we reevaluate the fundamental layer-based structure of containers. We identify that the proliferation of redundant files and operations within image layers hinders efficient container deployment. Drawing upon the crucial insight of enhancing layer reuse and extracting benefits from it, we introduce BREAK, a holistic approach centered on layer structure throughout the entire container deployment pipeline, ensuring backward compatibility. BREAK refactors image layers and proposes an edge-oriented cache solution to enable ubiquitous and shared layers. Moreover, it addresses the complete deployment pipeline by introducing a customized scheduler and a tailored storage driver. Our results demonstrate that BREAK accelerates the deployment process by up to 2.1× and reduces redundant image size by up to 3.11× compared to state-of-the-art approaches.

***Index Terms*—Container deployment, edge clouds, kubernetes**

I. Introduction

Container engines, such as Docker [1] and containerd [2], have shown their great advantages in standardization, user-friendliness, and low overhead [3], [4]. The container image, as a template for a container, is constructed by a series of layers. Users can package their applications as container by a standard way and upload them to a registry (e.g,. Docker hub [5]) for storage or sharing.

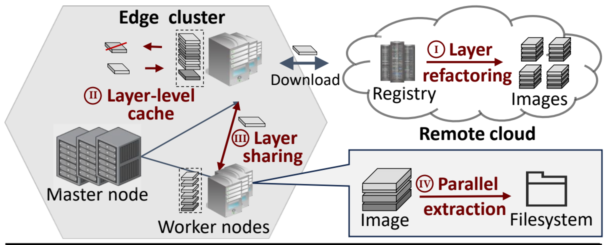
Containers need to rely on Container Orchestration Platforms (COPs) in clustered applications. COPs such as Kubernetes (K8s) [6], Docker Swarm [7], Apache Mesos (with Marathon) [8], are born to better manage the lifecycle of large-scale containers [9], automating configuration, placement (scheduling), scaling, etc. Among them, K8s is dominant, with 96% of organizations using or evaluating it [10].

While container technologies were initially conceived for cloud data centers [11], they are progressively gaining traction in edge computing [3], [12], [13]. These smaller, geographically distributed edge clouds are frequently organized by COPs as edge clusters within a particular region. Analogous to cloud data centers, edge clouds employ industry-standard hardware capable of running virtual machines and containers, thereby providing containerized services to end-users [14]. Due to proximity to users, these services reap benefits of reduced latency, power consumption, and bandwidth utilization [15]. As the edge accommodates a growing number of latency-sensitive services, the application deployment duration directly influences the quality of service [16], [17]. Sluggish deployments with elevated latencies may breach responsiveness Service-Level Agreements (SLAs) [3], [4], [18]. Simultaneously, redundancies among container images lead to expensive transfers, storage, and deployments [4], [18], [19].

Unique characteristics of edge environments pose challenges for container deployment. (i) Edge clouds may experience **prolonged container image retrieval times** from remote registries due to high-latency upstream links and bandwidth limitations [3], [13]. (ii) Oscillating network performance and heterogeneous resources [17], [20] further **complicate container placement** in geographically distributed edge clouds, impeding deployment. (iii) Given the resource limitations inherent to edge clouds, cache strategies that rely on storing entire container images are not cost-effective due to the **high storage overhead** they entail.

Prior work, mainly designed for cloud computing [13], [21], [22], is not well-suited for efficient deployment in edge clouds. These approaches can be categorized into two types. The first type, exemplified by FaaSNet [21] and Wharf [23], involves transferring container images on a large scale at default network settings, which are not suitable for bandwidth-constrained edge environments. The second type of solution is on-demand downloading or lazy-pulling [3], [24], but most of them cannot adapt to the unstable network latency characteristics of edge networks [3]. Some of these advanced efforts, such as Starlight [3], require a specific addressable image format, which necessitates additional format conversions from the current standard images, and such formats cannot be stored

Thanks to PPIO Cloud Computing (Shanghai) Co., Ltd. for providing the system platform and original logs. Xiaofei Wang was partially supported by the NSFC (Grant Nos. 62072332 and 62002260) and in part by the Tianjin Xinchuang Haihe Lab (Grant No. 22HHXCJC00002). Chenren Xu was supported by the NSFC (Grant Nos. 62022005, 62272010, and 62061146001). Qiao Xiang was supported by the NSFC (Grant No. 62022005), the MOE of China Award (Grant No. 2021FNA02008), and NSF-Fujian-China (Grant No. 2021J05003). Corresponding author: Xiaofei Wang.



Challenge 1: How to make image layers highly reusable?	
Ⓘ Layer refactoring	Low-Redundancy Image Refactoring (§III-B)
Challenge 2: How to make cache solution suited for edges?	
Ⓘⓘ Layer-level cache	Proactive Layer-Level Cache Pre-Fetching (§III-C)
Challenge 3: How to avoid downloading images remotely?	
Ⓘⓘⓘ Layer sharing	K8s Scheduler for Sharing Cache (§III-D)
Challenge 4: How to accelerate the image extraction process?	
Ⓘⓥ Parallel extraction	Asynchronous Parallel Extraction (§III-E)

Fig. 1. The challenges of efficiently deploying containers lies in four optimization dimensions around the layer structure.

in standard repositories. Furthermore, all these options do not account for a complete deployment pipeline for containers.

Container pulls, encompassing download and extraction operations, contribute to almost 80% of container deployment time [25]. We posit that the *primary hindrance* to efficient container deployment is the **multiple redundant operations such as storage, download, extraction** performed on individual files constituting a container image, resulting in deployment inefficiencies. The root cause of this phenomenon lies in the large number of redundant files hidden between images layers and the inefficient reuse of image layers [4], [26], [27].

In this paper, we propose a holistic approach to address the challenges of container deployment in distributed edge clusters. **Our key insight centers around enhancing image layer reuse and maximizing its benefits.** Importantly, we aim to improve the container itself without introducing potentially incompatible designs.

The effectiveness of our approach relies on tackling four significant challenges (see Figure 1). The first challenge is *how to make image layers highly reusable.* At present, most solutions struggle to balance de-duplication efficiency with compatibility, with some efforts focused solely on container registries. We perform image **layer refactoring**, eliminating redundant files between different layers. This enables container engines to reuse these consistent layers, thereby avoiding redundant storage, downloads, and extraction operations across the whole pipeline.

Second, *how to make cache solution suited for edges?* Edge resources are limited and heterogeneous, necessitating that both transmission and maintenance at the edge should be precise and flexible. Given the high-heat trend in container deployment (Figure 2), we utilize **layer-level caching** with an optimization policy to pre-fetch popular layers.

Next, *how to avoid high cost remote downloads?* Current K8s schedulers lack the capability for layer-level caching and are not cognizant of network performance. Given the limitations of edge networks to download images from remote clouds, we implement a customized K8s scheduler that facilitates collaborative deployment through **layer sharing** among geo-nearby edge clouds. This approach enables the retrieval

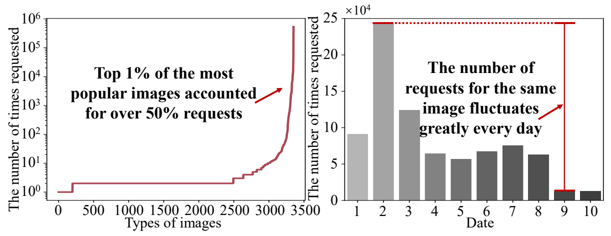


Fig. 2. Distribution of (a) the request number of different image types (left) and (b) the change in request number for an image over 10 days (right) in the IBM data [28]. The popularity-varied images serve as evidence to some extent for the necessity of caching and cache optimization schemes.

of only missing parts from the cloud.

The final challenge is *how to accelerate the image extraction process.* The current extracting operation in the image layer can slow down the deployment speed. However, to our knowledge, contemporary efforts have yet to address the inherent sequential and synchronous nature of these operations with compatible optimizations. To overcome this, we introduce a storage driver that allows the downloading and extraction of image layers **in parallel**.

We term these optimized designs as the BREAK approach, signifying **B**oosting efficient container deployment through layer **R**efactoring, **E**dge layer-level cache prefetching, **A**daptive layer sharing, and quic**K** layer extracting. In summary, our contributions are the following:

- We design an image refactoring solution which is backwards compatible with current container engines and standard registries. It optimizes and preserves the convenient stack-of-layers structure of images (§III-B).
- We propose a distributed, layer-level cache solution for layer pre-fetching, enabling cooperative container deployment by facilitating image layer transfer among geographically nearby edge clouds (§III-C).
- We develop a customized K8s scheduler which additionally considers network performance, disk space, and image layer caches to make appropriate container placements with layer sharing (§III-D).
- We identify the issues associated with current image extraction methods and propose a storage driver that enables parallel extraction of image layers, while eliminating redundant operations (§III-E).

We evaluate BREAK by employing 17 commonly-used container images, under various scenarios. Our findings indicate that BREAK performs 2.1× faster than the current state-of-the-art implementation, on average, while reducing redundant image size by up to 3.11×. While BREAK is specifically designed for edge scenarios, its backward-compatible features, such as image refactoring and extraction acceleration, can also prove beneficial for cloud computing.

II. Related Work

Container registry. Zhao et al. [29] proposes a solution for file-level deduplication in the registry after decompressing the layers, which involves hiding the overhead caused by reconstructing the compressed layers using a content-aware cache. Li et al. [26] argue that Docker CAS is limited because

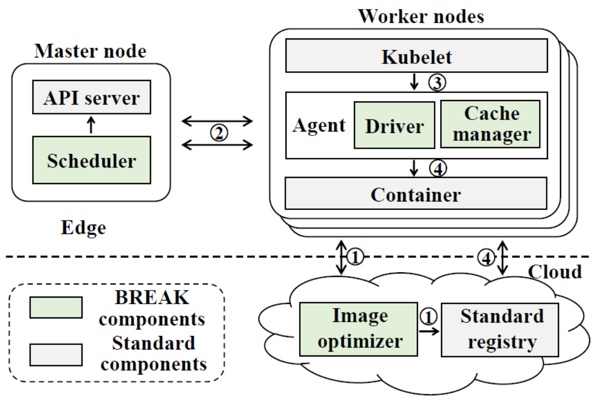


Fig. 3. The architecture of BREAK.

of layer-reuse in not sufficient and propose a reconstruction algorithm for Docker images in simulation. BREAK removes redundancy through container image refactoring, providing benefits not just limited to registry optimization, but also to other deployment processes. As BREAK's refactoring is based on the standard image specification, it has been proven to be fully compatible with container engines and registries.

On-demand downloading. Slacker [24] utilizes NFS to enable on-demand downloading of required data when containers are launched. Nydus [30] employs a file system to pull the data in chunk granularity, potentially leading to a degradation of native I/O performance. DADI [18] employs on-demand fetching at the block level, but requires a customized image format and registry. eStargz [31] and Starlight [3] use specialized filesystems and snapshotters to facilitate lazy data retrieval; however, their images require a complete conversion before use. In contrast, BREAK is designed based on existing container I/O stack, making it fully compatible.

P2P transmission. Existing solutions for optimizing container deployment, such as Wharf [23] and Shifter [32], offer client-side caching to share image. FaaSNet [21] accelerates Function-as-a-Service provisioning in datacenters using a tree of workers, while Dragonfly [33] and Krakenn [34] employ P2P approaches to reduce registry load in single datacenter settings. These solutions are designed primarily for cloud datacenters and do not address edge-specific challenges such as limited bandwidth and resource constraints. BREAK utilizes lightweight layer-level caching and P2P interactions among nearby edge clouds to cater to the unique characteristics of edge clouds.

III. Design Details

A. BREAK Architecture

Figure 3 illustrates the architecture of BREAK which comprised of three basic roles: *cloud (server registry)*, *master node* and *worker node*. A *K8s edge cloud cluster* typically consists of several geo-nearby nodes: one *master node* (at least) and a number of *worker nodes*. The former is the controller and the latter is used to deploy containers. Next, we will describe the main components in BREAK at a high level.

An *image optimizer* is primarily responsible for refactoring container images. It will periodically restructure the container images in the image registry by detecting redundancy in order to improve the reuse of image layers.

A *scheduler* is mainly responsible for: (i) determining the specific deployment node for the container through calculation and issuing the deployment event to that node, and (ii) obtaining a collaborative P2P transmission scheme for the current deployment within the cluster during the scheduling process, and notifying all worker nodes.

An *agent* is responsible for container starting and cache optimization. It mainly consists of (i) a *driver* component, which is used to asynchronously extract local image layers to start the container as quickly as possible, and (ii) a *cache manager*, which optimizes the caching of image layers on the local node to improve cache hit rates.

BREAK workflow. Firstly, the image optimizer refactors the container images in the registry to improve the reuse rate of image layers, thereby accelerating container deployment ①. This refactoring is periodic and triggered when the redundant file size of a container image in the registry exceeds a threshold. Both metadata files and image layer contents are modified, and a temporary copy is created to avoid interruption of image pulling during the refactoring of a specific image. Additionally, it synchronizes the metadata of the latest involved containers to the edge cloud and clears invalid caches ①. When the scheduler's watcher detects a new deployment event, it retrieves the resource and cache information of the cluster from the API server and local cache data center, and calculates the worker node on which the container should be deployed, thus completing the scheduling process ②. Meanwhile, based on the distribution of the cluster cache, the scheduler determines the P2P transmission scheme for image layer to accelerate image download, which is then notified to the worker nodes ②. The kubelet on the worker node receives the deployment event and notifies the agent ③. The agent writes the locally cached layers into the corresponding menu of the container filesystem through the driver, while asynchronously and in parallel extracts subsequent image layers from other worker nodes or registry in the cloud ④. In this process, the cache manager replaces the local cache in parallel. When all image layers are extracted, the container is started through runc (a standard runtime).

B. Low-Redundancy Image Refactoring

Each image layer represents an instruction in the image's Dockerfile and can only be shared if two layers are identical. Despite the current container solution, numerous redundant files remain unshared. The goal of image refactoring is to improve layer-level reuse capabilities offered by mainstream solutions, and achieve close to file-level reuse.

The overall workflow of image refactoring is shown in Figure 4: **Step 1:** BREAK generates file metadata for each image based on its initial structure, which includes basic information about each file. Using this metadata, BREAK then creates an image-level merged view by merging the image layers. **Step 2:** BREAK identifies redundant files between images using the merged views. **Step 3:** Based on the redundancy information,

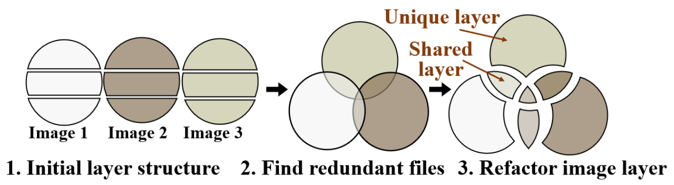


Fig. 4. Overview of container image refactoring.

BREAK divides the files into unique and shared layers. To optimize the process, Steps 1 and 2 are inferred based on the file metadata and the actual file manipulation occurs only after determination in Step 3. The following will be combined with the example of Figure 5 for a detailed description.

Step 1: Generating file metadata and merged view. In this step, BREAK generates file metadata for each image by parallelly iterating through all the files in the image and collecting the *path*, *name*, *size*, and *hash value (SHA 256)*. Then, it creates a merged view of the image by merging the initial layers according to the following methods: (i) If the files and folders in the lower layer have different paths compared to the upper layer, they are merged into the upper layer, as demonstrated by the *file x* in Figure 5 when *image 1* is merged based on *layer 1* and *layer 2*. (ii) If the paths and names of the files and folders in the lower layer are the same as the upper layer, the files and folders in the upper layer overwrite the lower layer, regardless of their contents, as demonstrated by *file /a/c/f* in Figure 5. (iii) BREAK deletes files and folders marked by "whiteouts" [35], such as *.wh..wh..opq* (hide all children) and *.wh.z* (hide *file z*) in Figure5.

Step 2: Determine the shareability of each file based on the merged view. First, a global key-value table is generated by iterating through the merged view of all images in parallel. The key of the table is the hash value of a file and its value is the list of images that contain that file. Files with the same value are candidates for sharing and are grouped into a shared layer, such as *file f* and *file i* in Figure 5. Files that are only contained in one image remain in the unique layer, such as *file x* and *file q* in Figure 5. Finally, the new layer structure of each image can be inferred based on the file metadata.

Step 3: Refactoring the new layer structure. To optimize efficiency, BREAK sets two thresholds (user can customize them as needed): (i) Layer creation threshold λ_l. To avoid excessive layer division, a shared layer is created only if its size s_l exceeds the threshold, i.e., $s_l \geq \lambda_l$; otherwise, the file remains in the unique layer. (ii) Trade-off threshold λ_a. Considering the image refactoring cost, the size of the added reusable files s_a needs to be calculated. If $s_a \geq \lambda_a$, the refactoring process is executed; otherwise, it is abandoned. If executed, the files selected in Step 2 are processed as in Figure 5. For example, *file f* and *file i* are moved to the shared path */a/z/* and their original locations are replaced with symbolic links pointing to the files in */a/z/*.

In addition, the native hierarchy faces significant challenges for images when updating versions [3]. When a new version of an image is released, typically only a few specific files are modified, while the majority of files from the operating system,

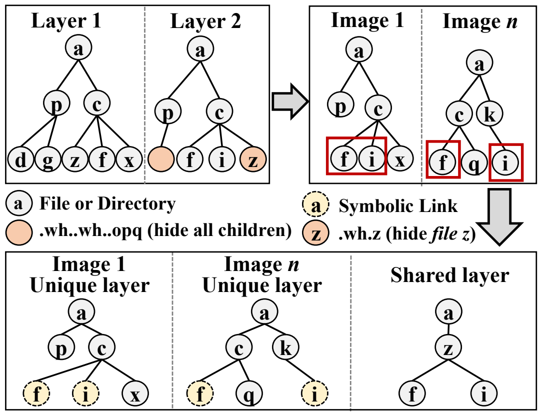


Fig. 5. Case study of container image refactoring.

runtime environment, etc. remain unchanged. This results in high redundancy between old and new versions. Due to the ability of image refactoring to reorganize the files contained in each layer, BREAK can move the modified files out of the shared layer while maintaining its shareability. In this regard, image refactoring can achieve an almost incremental update capability for image version iteration, which we will verify and quantify through experiments.

C. Proactive Layer-Level Cache Pre-Fetching

BREAK is designed to support distributed layer-level cache sharing and optimization, incorporating several key considerations. Firstly, container images exhibit dynamic heat distribution over time, making caching and optimization a practical and rational approach. Secondly, the stack-of-layers structure represents the prevailing standard for container images in the container I/O stack. Thirdly, caching more fine-grained popular layers (e.g., the base layer centOS) rather than an entire full image can lead to increased efficiency, particularly in resource-constrained edge cloud environments. Fourthly, the utilization of a layer-level cache allows for the full realization of the potential of layers after refactoring, resulting in improved layer reuse. Given the heterogeneous nature of edge clouds, users can set the caching space size when initiating the custom scheduler by applying K8s scheduler's YAML configuration.

Cache separation. In BREAK, image files are cached in a K8s cluster by separating them into two stores. Firstly, the master node, which is responsible for decision-making, caches image configuration metadata in a *metadata store*. Secondly, the compressed layer contents are cached in a *layer cache store* of worker nodes. These caches are organized using a menu tree structure. To ensure consistency, worker nodes periodically synchronize local cache information with the master node through HTTP. Upon starting or restarting BREAK, the local menu tree in each worker node is re-read to confirm consistency with the cache information.

Cache optimization. BREAK's caching algorithm, called ARC-LB (layer-based), is implemented based on the Adaptive Replacement Cache (ARC) algorithm [36]. It takes into consideration the size attribute of each image layer and ensures that the total size of the replaced layer is greater than or equal to the replacement layer during cache replacement.

ARC-LB keeps track of both recently and frequently used layers and adapts to changing access patterns. Initially, the cache space in each edge cloud gradually fills up with image layers pulled from the registry. Once the cache space is full, more frequently and recently used layers are prioritized as caches, and some of the used layers are replaced to free up space. [1] If a layer cache hits a container deployment event, the replacement and update of that layer cache will be postponed to prevent unnecessary additional pulling operations.

Collaborative deployment protocol. In BREAK, shared caches are transferred using a P2P approach among geo-nearby edge clouds, [2] enabling cooperative container deployment. To coordinate HTTP communication among the edge clouds and the registry, we design a deployment protocol shown in Figure 6. When a deployment request is invoked, the master node first issues the deployment event to the binding node selected as the host node for deployment (step I-1). The *Kubelet* in the binding node polls the image service (in containerd) to check if the requested image exists. Meanwhile, the master node sends the BREAK deployment manifest (BDM), which is the collaborative transfer solution, to the edge clouds (step I-2.1). The binding node also receives the image metadata for deployment (step I-2.2). According to the BDMs, peers send layers to the binding node (step I-3.1). If necessary, the binding node fetches layers from the registry (step I-3.2). Finally, the binding node sends back the deployment result and cache update to the master node (step I-4).

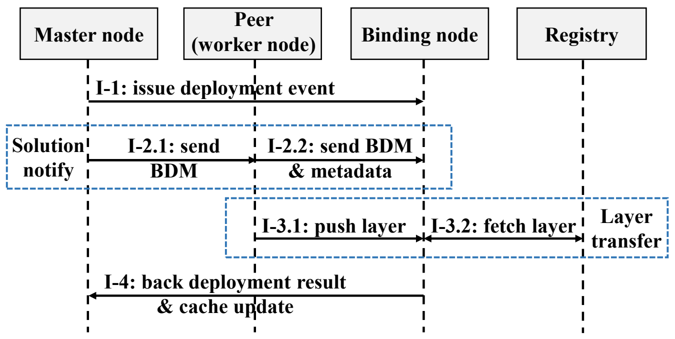


Fig. 6. Collaborative deployment protocol.

D. K8s Scheduler for Sharing Cache

As shown in Figure 7, we take advantage of the prior open-source practice [41] to implement the customized K8s scheduler, which comprises of a scheduling module for container placement and an orchestration module for generating collaborative transfer solutions.

Scheduling module. In K8s, container deployment events are triggered through the original CLI command by users, e.g., "kubectl apply service.yaml". Once the event is created, the *K8s API Server* notifies BREAK's customized scheduler for container scheduling. The native scheduling stream follows four steps: sorting, filtering, scoring, and binding. It relies on a set of plugins, and the node with the **highest score** is selected as the *binding edge node* for deploying in one turn.

[1] This is different from K8s default policy (i.e., garbage-collection [37]) which only focuses on local disk space without a clear cache definition.

[2] Our analysis, conducted on 10,000 edge clouds in the wild [38]–[40], reveals that geo-nearby edge clouds offer approximately three times the upstream bandwidth compared to remote ones, presenting additional potential for utilization.

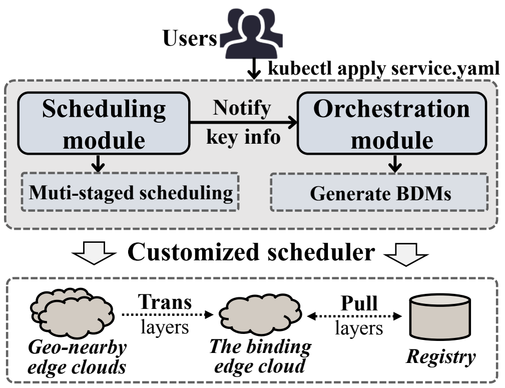


Fig. 7. The workflow of BREAK's scheduler.

In the scheduling module, we retain the primary multi-staged design of the native scheduler. However, we introduce modifications and extensions to four plugins for the scheduler: *(i) Network performance*, *(ii) Layer locality*, *(iii) Resources balanced allocation* and *(iv) Least requested priority*. All these plugins are assigned a default weight of 1 and can be adjusted by users as needed.

The *Network performance plugin* is aimed at making the scheduler network-aware. It calculates the network performance score $P_{i,net}$ of node n_i (where n represents the total number of edge nodes in the K8s cluster) based on the upstream bandwidth $b_{i,avg}$ and round-trip time $r_{i,avg}$ obtained from BREAK's exclusive measurement pod. Let $v_{i,b}$ and $v_{i,r}$ denote the ranking of $b_{i,avg}$ and $r_{i,avg}$ in the clusters, respectively, and λ_n be the constant factor for controlling the score scope. Therefore, $P_{i,net}$ can be derived using (1) as follows:

$$P_{i,net} = \lambda_n / n \cdot (2n - v_{i,b} - v_{i,r} + 2) \tag{1}$$

For edge nodes, disk over-occupation can significantly impact the node's stability. Therefore, we have introduced the *ephemeral storage* metric, which considers container storage usage, into both the *Resources balanced allocation* and *Least requested priority plugins*. Let $cpu_{i,o}$ and $cpu_{i,c}$ represent the occupied CPU and the capacity CPU of edge node n_i, respectively, and $ram_{i,o}$, $ram_{i,c}$ represent the memory resources, and $disk_{i,o}$, $disk_{i,c}$ represent the disk storage. Consequently, we define the score of the Resources balanced allocation plugin as $P_{i,bal}$ in (2), and $P_{i,pri}$, representing the score of the Least requested priority plugin, is shown in (3). Similarly, λ_b and λ_p are the constant factors for scope control.

$$\begin{aligned} P_{i,bal} = \lambda_b - \Bigg(\left| \frac{cpu_{i,o}}{cpu_{i,c}} - \frac{ram_{i,o}}{ram_{i,c}} \right| + \left| \frac{cpu_{i,o}}{cpu_{i,c}} - \frac{disk_{i,o}}{disk_{i,c}} \right| \\ + \left| \frac{ram_{i,o}}{ram_{i,c}} - \frac{disk_{i,o}}{disk_{i,c}} \right| / 3 \Bigg) \cdot \lambda_b \end{aligned} \tag{2}$$

$$\begin{aligned} P_{i,pri} = \Bigg(\frac{cpu_{i,c} - cpu_{i,o}}{cpu_{i,c}} + \frac{ram_{i,c} - ram_{i,o}}{2 \cdot ram_{i,c}} \\ + \frac{disk_{i,c} - disk_{i,o}}{10 \cdot disk_{i,c}} \Bigg) \cdot \lambda_p \end{aligned} \tag{3}$$

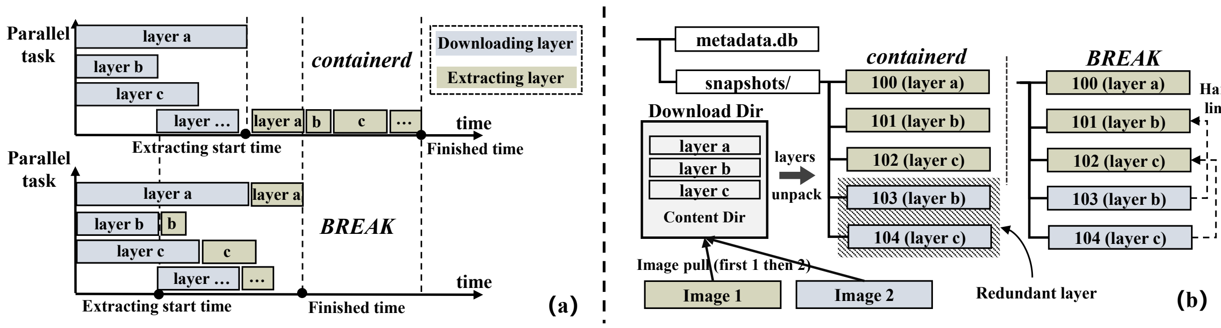


Fig. 8. Comparison of containerd and BREAK's layer extraction. In containerd, layer extraction operations are dependent on the completion of the download operation, resulting in a sequential process. In contrast, BREAK enables concurrent downloads and extractions, effectively decoupling the sequential loading of layers. Additionally, BREAK facilitates hard-linking of identical layers (read-only layer sections), leading to a reduction in redundant extraction operations.

To better accommodate BREAK's layer-level operation, we have replaced the native *Image locality* plugin with a more fine-grained *Layer locality* plugin, which considers layer cache. The score $P_{i,lay} = p_{i,hit} + \eta \cdot p_{i,ava}$ describes the metrics of the proportion of hitting layer size $p_{i,hit}$ and the current available cache space $p_{i,ava}$ of node n_i. These can be calculated by:

$$p_{i,hit} = \lambda_l \cdot (q_i - \theta_{min}) \ / \ (\theta_{max} - \theta_{min}) \tag{4}$$

$$p_{i,ava} = (cache_{i,c} - cache_{i,o}) \ / \ cache_{i,c} \tag{5}$$

where θ_{min} and θ_{max} are thresholds of the sum of hitting layer caches $p_{i,hit}$ (the value of q_i is reassigned when it exceeds the thresholds), and λ_l is the constant factor. Here, $cache_{i,c}$ and $cache_{i,o}$ represent the capacity and the occupied cache space of edge node n_i. Note that $p_{i,ava}$ is designed to avoid node heating problems [42], enabling containers to be more evenly distributed across the cluster (meeting robustness requirements) and fully utilizing each node's cache space. In general, the larger the value of $P_{i,lay}$, the more cache layers are hit, and the richer the cache space becomes.

Based on the scores from each plugin, we calculate the final score P_i of node n_i as follows:

$$\begin{aligned} P_i &= \sum_{k \in \mathcal{K}} \omega_k \cdot P_{i,k} \\ &= \omega_{\text{net}} \cdot P_{i,\text{net}} + \omega_{\text{bal}} \cdot P_{i,\text{bal}} + \omega_{\text{pri}} \cdot P_{i,\text{pri}} + \omega_{\text{lay}} \cdot P_{i,\text{lay}} + \cdots \end{aligned} \tag{6}$$

where $\mathcal{K}$ is the set of plugins, and ω_k represents the corresponding weight of plugin k. BREAK's scheduler can be deployed as a containerized component in K8s through configuration, allowing users to customize the weight for each plugin by changing the factors of the scheduler module in the configuration.[3]

The node with the highest score will be selected as the binding node to which the deployment is issued. Additionally, the module notifies the orchestration module about the binding node and the network performance of cluster nodes. Subsequently, the deployment event is issued to the *Kubelet* of the binding node.

Orchestration module. The orchestration module is designed based on the collaborative deployment protocol (§III-C). When a deployment event occurs, the orchestration module first confirms whether the container is cached in the edge cloud cluster based on the application YAML configuration submitted by users. A cooperative transfer solution for deployment is generated based on (i) the metadata from the metadata store or pulling from remote registry and (ii) the information notified by the scheduling module. In particular, the orchestration module creates and issues BDMs, which include transfer tasks for each edge node. The module gives preference to the node with good network performance for layer transfer.

For instance, if the binding node "node1" needs "layer1" to rebuild the container image, the module informs "node2" – an edge cloud in the cluster who caches "layer1" with good network performance – to push the layer to "node1". If the required "layer1" is not available in the cluster or if all other node networks are performing poorly, "node1" will be informed to request it from the registry server via the registry REST API. To avoid costly round-trip requests, BDMs are pushed by the master node rather than the binding node. For practical purposes, the default concurrent upload connections setting number is 3, which can be changed as needed.

E. Asynchronous Parallel Extraction

Extraction of image content from a tarball into the local filesystem is facilitated by a storage driver. However, current drivers such as Docker's Overlay2 and containerd's snapshotter have limitations that hinder their efficiency.

Firstly, image layers must be extracted in the order of the image structure, and in the case of containerd's snapshotter, extraction is delayed until after the download completes. Secondly, the layer content cannot be efficiently reused because these drivers calculate the *chainID* [43] in real-time, which is a SHA256 value that combines one layer's content and all its parent layers to indicate the layer's unique identity in container. This process is time-consuming for image layers that will be mounted as read-only levels, leading to redundant extraction operations.

Design expectation. To enhance the extraction performance, we propose the BREAK driver, which is built on the Docker graph-driver plugin SDK and Docker Overlay2 driver. It preserves the advantages of Docker Overlay2, such as the ability to mount layers, while improving the extraction process.

As shown in Figure 8, our design aims to initiate image

[3] We have retained the K8s weighted approach to calculating scores, making extensions to the native plugin backward compatible. Empirically, we take the K8s default value of 10 for the λ constant factor.

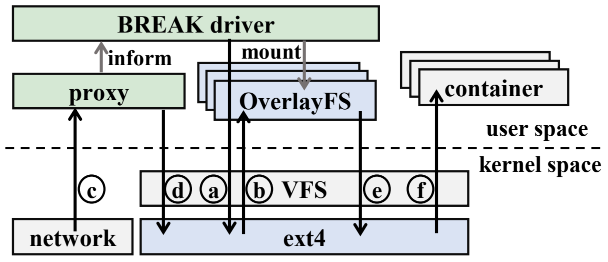


Fig. 9. BREAK's storage driver workflow.

extraction earlier, allowing it to overlap with the download process. We achieve this by extracting each layer asynchronously and concurrently. Furthermore, we enable the reuse of layer content, which is achieved by calculating the *chainID* in advance, rather than in real-time during extraction.

Workflow analysis. To better illustrate the advantages of our BREAK driver in image extraction, we provide a specific description in Figure 9. To start the container quickly, once the image configuration metadata is received on the binding node (determined by the scheduler), BREAK driver initializes a series of filesystem menus for this container image based on the metadata files ⓐⓑ. This is different from the original Overlay2 driver as BREAK driver decouples image metadata writes from image layer content writes, allowing the extraction of images to be done in any order without disrupting the union mounting of the image layers, which still considers the original order of layers obtained from the configuration metadata.

BREAK driver first writes the metadata, including imagedb, distribution, layerdb, repositories, etc., and then writes the contents of the existing layers to the filesystem menu. This is designed because the image configuration metadata, which is probably only a few KB in size, can be received faster than the layer contents. The required layer contents, in the best scenario, are already in the local filesystem (e.g., ext4) as maintained cache, otherwise they will be downloaded by the proxy (an auxiliary component designed for BREAK dirver). Once the proxy has downloaded and decompressed new layer tarballs ⓒⓓ, it informs the BREAK driver to move these read-only layers to the corresponding lower directories ⓐⓑ.

Although BREAK driver still retains the concept of chainID, it uses diffID (SHA256 value of layer uncompressed content) to retrieve the identity of layer contents in directories. This helps to reuse layers with different chainIDs but consistent content by creating hard links, avoiding meaningless extraction. All these files in OverlayFS can be returned to the container instance ⓔⓕ. If any changes are made to files metadata and content by the container instance, these files will be copied from the read-only layer to the read-write layer, where subsequent requests are forwarded.

Lazy-pulling option. For many containerized applications that do not need to read all layers immediately, BREAK's extraction mechanism supports them with an option to fetch in a lazy way. Similarly like work [44], by preparing a boot layer which contains the minimum files needed for container startup, BREAK can also start the container early potentially and extracts other requested layers asynchronously and in parallel while blocking files access until notified of completion.

IV. Evaluation

In this section, we conduct a comprehensive evaluation of BREAK, focusing on the (i) effectiveness of the refactoring solution (§IV-B), the (ii) performance of the layer-level cache solution (§IV-C), the (iii) comparison of the customized K8s scheduler (§IV-D), the (iv) improvements brought by the storage driver (§IV-E), and (v) BREAK's adaptability under various edge network environments (§IV-F).

A. Experimental Setup

Testbed setup. Our testbed consists of four edge cloud clusters (each with 1 master node and 4 worker nodes) and a registry server. Each worker node is equipped with 2 cores (vCPUs, 2.20GHz Intel Xeon E5-2630) and 4GB RAM, while the master node and registry server are configured with 4 vCPUs and 8GB RAM. The release of K8s v1.24.10 is deployed on clusters and Docker Registry2.0 v2.8.1 is chosen as the standard registry on the registry server. All the machines run Ubuntu 20.04.3 LTS. To control the bandwidth and Round-Trip Time (RTT) of the edge cloud cluster and registry server, we use the Linux Traffic Control (TC) tool [45]. The network bandwidth ceiling is limited to 500Mbps and the RTT is set to 15ms within one edge cloud cluster.

Containers and workloads. In our evaluation of BREAK, we consider a set of 17 popular official images from the Docker Hub [5] with a total size of 5.96GB. To ensure real-world relevance, we use a workload dataset from IBM [28] for our experiments. Specifically, the "timestamp" in the dataset is taken as the request arrival time, and the "http.request.uri" is considered as the container type.

B. Refactoring Effectiveness

We first verify the effect of image refactoring on version updates based on 17 popular official images from the Docker Hub. The average of shareable files (files can be reused) is only 5.71% (see Table I). It means that in version updates without image refactoring, most image layers need to be reacquired.

Image name	1.python	2.golang	3.openjdk	4.ubuntu	5.memcached
6.httpd	7.mysql	8.mariadb	9.redis	10.postgres	11.rabbitmq
12.registry	13.wordpress	14.ghost	15.node	16.flink	17.cassandra

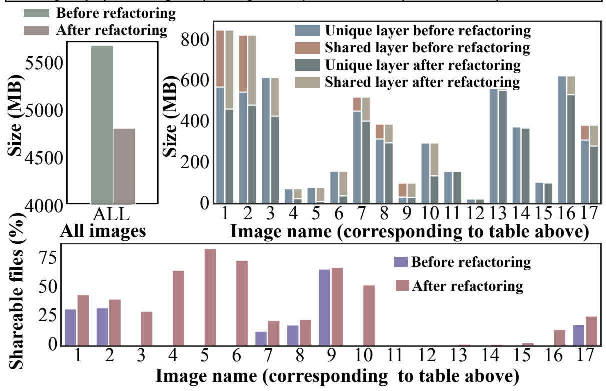


Fig. 10. Names of images involved (top), impact of refactoring on image size (middle), and percentage of shareable files (bottom).

TABLE I
ANALYSIS ON THE PERCENTAGE OF SHAREABLE FILES FOR IMAGE VERSION UPDATES.

Image	Version	Before refactoring	After refactoring	Image	Version	Before refactoring	After refactoring
Python	3.9.3 → 3.9.4	0 %	97.54 %	Postgres	13.1 → 13.2	0 %	98.12 %
Golang	1.16.2 → 1.16.3	0 %	97.94 %	Rabbitmq	3.8.13 → 3.8.14	0 %	98.84 %
Openjdk	11.0.11-9-jdk → 11.0.12-jdk	0 %	98.62 %	Registry	2.7.0 → 2.7.1	0 %	98.97 %
Ubuntu	focal-20210401 → focal-20210416	0 %	98.93 %	Wordpress	php7.3-fpm → php7.4-fpm	0 %	98.30 %
Memcached	1.6.8 → 1.6.9	0 %	96.22 %	Ghost	3.42.5-alpine → 3.42.6-alpine	1.42 %	86.21 %
Httpd	2.4.41 → 2.4.43	0 %	97.05 %	Node	16.19-alpine3.16 → 16.19-alpine3.17	0 %	98.21 %
Mysql	8.0.23 → 8.0.24	24.79 %	99.23 %	Flink	1.12.3 → 1.12.4	0 %	99.08 %
Mariadb	10.5.8 → 10.5.9	0 %	98.96 %	Cassandra	3.11.9 → 3.11.10	0 %	97.80 %
Redis	6.2.1 → 6.2.2	70.85 %	97.01 %	**Average**	/	**5.71 %**	**97.47 %**

But in fact, a large part of these files still duplicate hidden in different layers. Under the current naive container mechanism, unfortunately, even a minor change to a single layer would cause the entire layer to lose the capability to share. In response, BREAK achieves near-incremental version update capability by optimizing the files contained in each layer, with an average shared file of 97.47%. This provides a solution to maintain compatibility while enabling incremental updates.

In addition to different versions of the same image, we also quantify the ability of image refactoring to remove redundancy between different images. In Figure 10, we compare the total storage size of 17 container images in the registry. After refactoring, BREAK effectively reduces the redundant size of images by up to 3.11×. Note that BREAK's container image refactoring not only reduces image storage space in the registry, but also benefits from a series of deployment processes such as download and load while retaining the container key design of the stack-of-layers structure.

C. Cache Analysis

Next, we analyze the behavior of BREAK's distributed shared cache. First, to get the performance of BREAK's cache algorithm ARC-LB, we show its hit ratio (including local hit ratio and edge hit ratio) compared to two popular cache algorithms LRU [46] and LFU [36] under different number of worker node in the edge cloud cluster in Figure 11 (a).

The local hit ratios of ARC-LB are the highest at 0.16, 0.24,

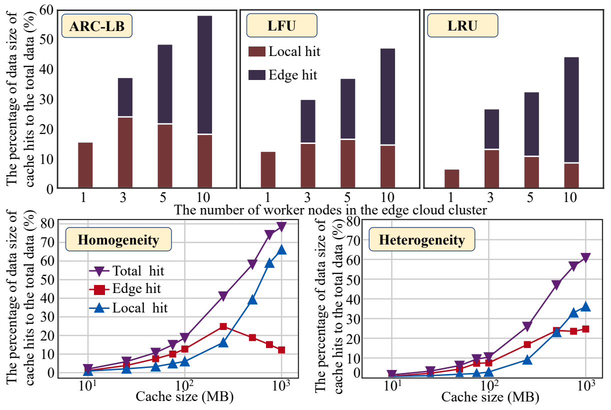


Fig. 11. Performance comparison of (a) different cache algorithms (top) and (b) different cache size (bottom).

0.22 and 0.18, respectively. This means that ARC-LB shows a better optimization ability. Compared to others, ARC-LB maintains two lists at the same time, an LFU list and an LRU list, and adaptively balances them to increase the hit ratio.

For different number of working nodes, the edge hit ratio (i.e., that image layer is cached on other worker nodes rather than the local node) is the highest for ARC-LB in most cases, indicating that the algorithm effectively optimizes the total cache distribution in the cluster.

BREAK supports custom size cache space set by the user for worker nodes. In Figure 11(b), we compare the ARC-LB hit ratio of a cluster (with 10 worker nodes) under cache homogeneous distribution (i.e., each node has the same cache size) with heterogeneous distribution (conforming to an equal distribution) under three different cache sizes. In general, the cache has a higher total hit rate in the homogeneous case. When the cache size is 1000M, it is 17.57% higher than the heterogeneous case.

In addition, we find that at a cache of 1000M, the local hit rate is much higher than the edge hit rate in the homogeneous case compared to the heterogeneous case. This implies that ARC-LB achieves better optimization of the cache for local nodes in the homogeneous case.

D. Scheduler Comparison

To verify the performance of BREAK's scheduler, we compare the scheduler with two state-of-the-art methods: the *K8s default scheduler* baseline [47] and *NetMARKS* [48]. The baseline considers node resource balance, complete image cache, etc. NetMARKS leverages Istio Service Mesh [49] to select the most bandwidth-rich nodes for container deployments. We set the cache size of worker nodes to vary from 0% to 25% of the workload dataset size (i.e., the percentage of top in Figures 12 and 14) and run the experiments in the BREAK architecture with only the scheduler replaced.

Figure 12 shows the performance of different schedulers in terms of deployment time. As the cache size increases, all deployment time decreases to some extent (shown as downward aggregation in the graph). However, NetMARKS shows the least significant change because it does not take into account node caching. Even though the baseline takes into account the cache situation, it is only full image aware

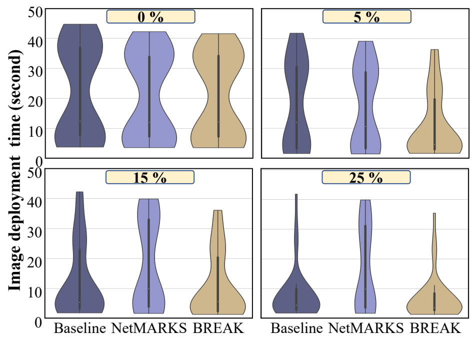


Fig. 12. Analysis of the deployment time distribution of images by different schedulers under different cache sizes.

and is not aware of the hotness layer of the cache. Compared to others, BREAK's scheduler takes into account not only the network conditions of the nodes, but also the more granular cache layer – it is implemented more like an enhanced version of baseline combined with NetMARKS. As a result, BREAK's scheduler achieves the fastest container deployment.

E. Extraction Performance

In our evaluation of image extraction, we consider 17 different images as depicted in Figure 10. As shown in Figure 13 (a), the average extraction time for these images using containerd is 2.941 seconds. Since the extraction phase in containerd needs to be executed sequentially layer by layer only after all the layers in the image have been downloaded, this significantly increases the time consumption.

In contrast, as shown in Figure 13 (b), the average extraction time for BREAK is 0.599 seconds, which is 4.9× faster than the native solution. BREAK enables disordered layer extraction, meaning that extraction can be performed as soon as the child layer has finished downloading, even if the parent layer is still downloading. Additionally, BREAK allows for parallel loading of different layers, enabling multiple layers to be extracted simultaneously, instead of sequentially.

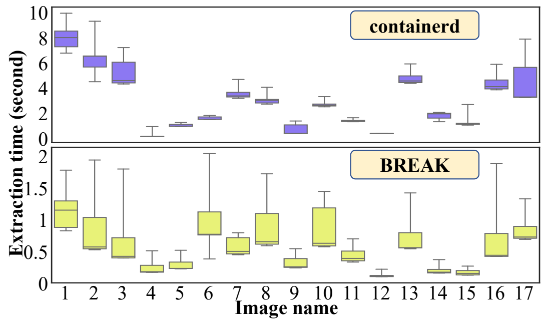


Fig. 13. Comparison of (a) containerd (top) and (b) BREAK (bottom) at image extraction time.

F. Deployment Time

The deployment time of containers is the main metric of interest, defined as the time elapsed from the issuance of the initial command to deploy a container. We monitor deployment status by periodically sending pod information requests to the *K8s API Server* every 200ms. Three different network environments are considered: (*i*) poor: network bandwidth with 50Mbps ceiling and 200ms RTT (from the edge cloud to the remote registry); (*ii*) moderate: network bandwidth with 100Mbps ceiling and 50ms RTT; (*iii*) excellent: network bandwidth with 500Mbps ceiling and 10ms RTT.

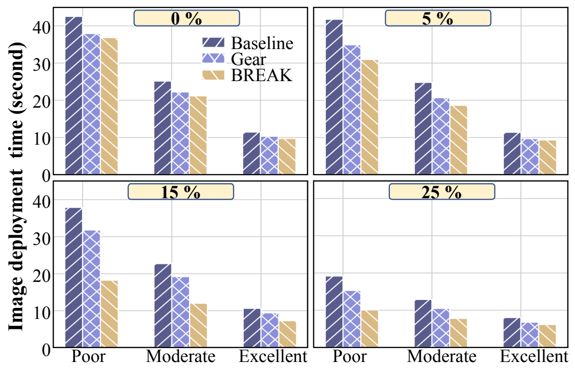


Fig. 14. Overall performance comparison regarding image deployment time.

We compare BREAK with two state-of-the-art approaches: the K8s v1.24.10 baseline (with containerd v1.6.4) and Gear [4]. Gear introduces a new image format and uses a file-based sharing mechanism to achieve efficient container deployment. We selected Gear for comparison as, similar to BREAK, it does not alter the container I/O stack and considers both the container image storage and pull scenarios, providing a relatively complete deployment process. However, since Gear does not integrate with K8s, its consideration of the full container deployment process is incomplete. To address this, we made the necessary modifications, such as adding a scheduling process using the default scheduler of K8s.

Figure 14 presents the average deployment time of containers as a function of network performance and cache size, based on the workload dataset described in Section IV-A. The results indicate that BREAK consistently demonstrates the fastest deployment time across all network performance and cache size scenarios. As the cache size increases, BREAK gains the most from caching, owing to its BREAK's (*i*) layer-level cache granularity, (*ii*) efficient cache optimization algorithms, and (*iii*) a custom scheduler that is aware of layer caches. BREAK over other solutions is particularly pronounced, with a deployment time that is 2.1× faster than the baseline and 1.7× faster than Gear. On average, BREAK is 1.4× faster than the baseline and 1.2× faster than Gear. It is worth noting that neither removing redundancy nor image caching alone is sufficient to achieve the level of performance demonstrated by BREAK, whose effectiveness stems from its overall design.

V. Conclusion

This paper presents BREAK, a holistic approach tailored to enhance container deployment in edge clouds. We introduced the nascent ideas of BREAK in [50], and this paper significantly extends and verifies those ideas with transformative advancements. BREAK incorporates various mechanisms and components that seamlessly integrate into the pipeline, ensuring backward compatibility. The results indicate that BREAK significantly accelerates the deployment process, achieving up to 2.1× faster speed, and effectively reduces redundant image size by up to 3.11× compared to state-of-the-art methods.

References

[1] "Docker: Accelerate how you build, share, and run modern applications," https://www.docker.com/.

[2] "Containd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability," https://containerd.io/.

[3] J. L. Chen, D. Liaqat, M. Gabel, and E. de Lara, "Starlight: Fast container provisioning on the edge and over the wan," in *USENIX Symposium on Networked Systems Design and Implementation*, 2022, pp. 35–50.

[4] H. Fan, S. Bian, S. Wu, S. Jiang, S. Ibrahim, and H. Jin, "Gear: Enable efficient container storage and deployment with a new image format," in *International Conference on Distributed Computing Systems*, 2021, pp. 115–125.

[5] "Build and ship any application anywhere," https://hub.docker.com/.

[6] "Kubernetes: Production-grade container scheduling and management," https://github.com/kubernetes/kubernetes.

[7] "Swarm mode overview," https://docs.docker.com/engine/swarm/.

[8] "Apache mesos," https://github.com/apache/mesos.

[9] P. Ambati and D. Irwin, "Optimizing the cost of executing mixed interactive and batch workloads on transient vms," *ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 2, pp. 1–24, 2019.

[10] "Cncf annual survey 2021," https://www.cncf.io/reports/cncf-annual-survey-2021/.

[11] "Cloud native computing foundation," https://cncf.io.

[12] B. Costa, J. Bachiega Jr, L. R. de Carvalho, and A. P. Araujo, "Orchestration in fog computing: A comprehensive survey," *ACM Computing Surveys*, vol. 55, no. 2, pp. 1–34, 2022.

[13] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling." in *USENIX Workshop on Hot Topics in Edge Computing*, 2020.

[14] P. Lai, Q. He, G. Cui, F. Chen, M. Abdelrazek, J. Grundy, J. Hosking, and Y. Yang, "Quality of experience-aware user allocation in edge computing systems: A potential game," in *IEEE International Conference on Distributed Computing Systems*, 2020, pp. 223–233.

[15] B. Varghese, E. De Lara, A. Y. Ding, C.-H. Hong, F. Bonomi, S. Dustdar, P. Harvey, P. Hewkin, W. Shi, M. Thiele *et al.*, "Revisiting the arguments for edge computing research," *IEEE Internet Computing*, vol. 25, no. 5, pp. 36–42, 2021.

[16] H. Lu, G. Xu, C. W. Sung, S. Mostafa, and Y. Wu, "A game theoretical balancing approach for offloaded tasks in edge datacenters," in *IEEE International Conference on Distributed Computing Systems*. IEEE, 2022, pp. 526–536.

[17] Y. Feng, S. Shen, M. Xu, Y. Ren, X. Wang, V. C. Leung, and W. Wang, "Tango: Harmonious management and scheduling for mixed services co-located among distributed edge-clouds," in *Proceedings of the 52nd International Conference on Parallel Processing*, 2023, pp. 595–604.

[18] H. Li, Y. Yuan, R. Du, K. Ma, L. Liu, and W. Hsu, "Dadi: Block-level image service for agile and elastic application deployment," in *USENIX Conference on Usenix Annual Technical Conference*, 2020, pp. 727–740.

[19] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt, "Large-scale analysis of the docker hub dataset," in *IEEE International Conference on Cluster Computing*, 2019, pp. 1–10.

[20] M. Xu, Z. Fu, X. Ma, L. Zhang, Y. Li, F. Qian, S. Wang, K. Li, J. Yang, and X. Liu, "From cloud to edge: a first look at public edge platforms," in *ACM Internet Measurement Conference*, 2021, pp. 37–53.

[21] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *USENIX Annual Technical Conference*, 2021.

[22] N. Zhao, V. Tarasov, A. Anwar, L. Rupprecht, D. Skourtis, A. Warke, M. Mohamed, and A. Butt, "Slimmer: Weight loss secrets for docker registries," in *IEEE International Conference on Cloud Computing*, 2019, pp. 517–519.

[23] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain, M. Mohamed, D. Skourtis, A. S. Warke, and D. Hildebrand, "Wharf: Sharing docker images in a distributed file system," in *ACM Symposium on Cloud Computing*, 2018, pp. 174–185.

[24] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in {*USENIX*} *Conference on File and Storage Technologies*, 2016, pp. 181–195.

[25] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *European Conference on Computer Systems*, 2015, pp. 1–17.

[26] S. Li, A. Zhou, X. Ma, M. Xu, and S. Wang, "Commutativity-guaranteed docker image reconstruction towards effective layer sharing," in *ACM Web Conference*, 2022, pp. 3358–3366.

[27] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. K. Paul, K. Chen, and A. R. Butt, "Large-scale analysis of docker images and performance implications for container storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 918–930, 2020.

[28] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig *et al.*, "Improving docker registry design based on production workload analysis," in {*USENIX*} *Conference on File and Storage Technologies*, 2018, pp. 265–278.

[29] N. Zhao, H. Albahar, S. Abraham, K. Chen, V. Tarasov, D. Skourtis, L. Rupprecht, A. Anwar, and A. R. Butt, "Duphunter: Flexible high-performance deduplication for docker registries," in *USENIX Annual Technical Conference*, 2020.

[30] "Nydus," https://github.com/dragonflyoss/image-service.

[31] D. Richie, *A hundred years of Japanese film: a concise history, with a selective guide to DVDs and videos*. Kodansha International, 2005.

[32] L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia, "Shifter: Containers for hpc," in *Journal of physics: Conference series*, vol. 898, no. 8. IOP Publishing, 2017, p. 082021.

[33] "What is dragonfly?" https://d7y.io/docs/.

[34] "kraken: A p2p-powered docker registry," https://github.com/uber/kraken.

[35] "Image layer filesystem changeset," https://github.com/opencontainers/image-spec/blob/main/layer.md#whiteouts.

[36] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache," in *Fast*, vol. 3, no. 2003, 2003, pp. 115–130.

[37] "Garbage collection," https://kubernetes.io/docs/concepts/architecture/garbage-collection/.

[38] S. Shen, Y. Feng, M. Xu, C. Zhang, X. Wang, W. Wang, and V. C. Leung, "A holistic qos view of crowdsourced edge cloud platform," in *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*. IEEE, 2023, pp. 01–10.

[39] Y. Feng, S. Shen, M. Xu, C. Zhang, X. Wang, X. Wang, W. Wang, and V. C. Leung, "A large-scale holistic measurement of crowdsourced edge cloud platform," *World Wide Web*, vol. 26, no. 5, pp. 3561–3584, 2023.

[40] "Ppio edge cloud," https://www.ppio.cn/.

[41] D. Haja, M. Szalay, B. Sonkoly, G. Pongracz, and L. Toka, "Sharpening kubernetes for the edge," in *ACM SIGCOMM Conference Posters and Demos*, 2019, pp. 136–137.

[42] "Node heating problem," https://oracle.github.io/weblogic-kubernetes-operator/faq/node-heating/.

[43] "Chainid," https://github.com/moby/moby/blob/v23.0.4/layer/layer.go#L195.

[44] S. Gotanda and T. Shinagawa, "Short paper: Highly compatible fast container startup with lazy layer pull," in *IEEE International Conference on Cloud Engineering*, 2021, pp. 53–59.

[45] W. Almesberger, "Linux traffic control-implementation overview," Tech. Rep., 1998.

[46] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *Acm Sigmod Record*, vol. 22, no. 2, pp. 297–306, 1993.

[47] "kube-scheduler," https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

[48] Ł. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh," in *IEEE Conference on Computer Communications*, 2021, pp. 1–9.

[49] "service-meshservice-mesh," https://istio.io/latest/about/service-mesh/https://istio.io/latest/about/service-mesh/.

[50] Y. Feng, S. Shen, C. Zhang, and X. Wang, "Quicklayer: A layer-stack-oriented accelerating middleware for fast deployment in edge clouds," in *Proceedings of the 7th Asia-Pacific Workshop on Networking*, 2023, pp. 74–80.