Commutativity-guaranteed Docker Image Reconstruction towards Effective Layer Sharing

Sisi Li
Beijing University of Posts and Telecommunications
Beijing, China
sisili@bupt.edu.cn

Ao Zhou*
Beijing University of Posts and Telecommunications
Beijing, China
aozhou@bupt.edu.cn

Xiao Ma
Beijing University of Posts and Telecommunications
Beijing, China
maxiao18@bupt.edu.cn

Mengwei Xu
Beijing University of Posts and Telecommunications
Beijing, China
mwx@bupt.edu.cn

Shangguang Wang
Beijing University of Posts and Telecommunications
Beijing, China
sgwang@bupt.edu.cn

ABSTRACT

Owing to the benefit of light weight, containers have become a promising enabler for cloud native computing. Container images composed of applications and dependencies support flexible service deployment and migration. Rapid adoption and integration of containers generate millions of images to be stored. Additionally, non-local images have to be frequently downloaded from the registry, resulting in huge amounts of traffic. Content Addressable Storage (CAS) has been adopted for saving storage and networking by enabling identical layers sharing across images. However, according to our measurements, the implication of CAS is significantly limited as layers are rarely fully identical in practice. In this paper, we propose to reconstruct the docker images to raise the number of identical layers and thereby reduce storage and network consumption. We explore the layered structure of images and define the commutativity of files to assure image validity. The image reconstruction is formulated as an integer nonlinear programming problem. Inspired by the observed similarity of layers, we design a similarity-aware online image reconstruction algorithm. Extensive evaluations are conducted to verify the performance of the proposed approach.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Human-centered computing** → *Ubiquitous and mobile computing*.

KEYWORDS

Containers, Container images, Docker, Docker hub

*Ao Zhou is the corresponding author.

ACM Reference Format:
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 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3485447.3512154

1 INTRODUCTION

Containers have revolutionized how web applications are deployed, including social networking, e-commerce, etc [17][24][27]. Compared with virtual machines (VMs) [23], containers are more lightweight and incur less overhead by sharing the operating system kernel [12]. Moreover, containers enable developers to focus more on application logic without worrying about the underlying infrastructure [19]. Owing to these advantages, containers are envisioned as the key technique to achieve cloud native vision [8].

The executable files and runtime dependencies of containers are packaged into images. Flexible sharing of containers is achieved by delivering images among machines. Images are structured in independent layers and each layer is a subset of files. All the images are stored in a centralized registry, which clients "push" new images to, and "pull" existing images from to start and run containers at hosts. With the development of containers and cloud computing technologies, the number of images is growing rapidly, consuming considerable storage and networking resources. For example, as the most popular image registry, Docker Hub [2] now stores more than 2 million public images and over 400 million private images, occupying roughly 1 PB storage. Moreover, the ubiquitous use of internet connected mobile devices aggravates the mobility and dynamism of services. Google starts an average of 7000 containers per second [14] for supporting the continuously changing requests. Frequent service deployment and migration can pose huge pressure on storage and networking.

Content-addressable storage (CAS) [29] has been employed to relieve the burden of storage and networking by reusing the identical layers among different images at the same client server. Layer-aware image caching [26] and service placement [14][13] have been investigated. However, as presented in our measurement (Section 3), CAS provides 38% storage saving while 35% of file storage is still redundant. The main reason is that the number of identical

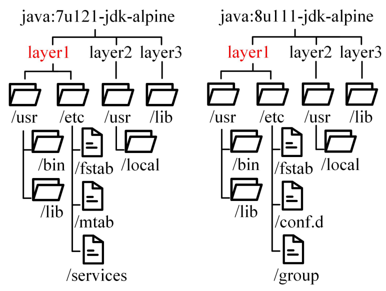


Figure 1: The layered structure of image *java*. Red font marks the "similar layers".

layers among images is quite limited. More than 90% of layers are unique, and only less than 1% of layers are shared by more than 25 images [33]. Therefore, the potential of layer sharing remains to be explored.

Based on layer sharing, deduplication at file granularity is leveraged to further save storage [32][34]. Redundant files are removed and only unique files are stored in the registry. The metadata file "recipe" records the files of each layer. However, the files would be reconstructed into layers according to "recipe" and then delivered to clients when requested. Clients are still required to download and store all the unique layers with redundant files. Therefore, these investigations help save storage for the registry yet cannot reduce the network traffic or clients' storage footprint.

While a minority of layers are shared among images, the enormous identical files lead to many similar layers [7]. Figure 1 presents partial layers and files of the image *java: 7u121-jdk-alpine* and *java: 8u111-jdk-alpine*, where the red font marks the similar layers. Most of the files across the two layers are duplicated, with only a few files distinct. According to our measurement on the similarity with 2,200 images, we observe that 26,042 layer pairs have more than 80% identical files. Moreover, it is unearthed that the layer pairs in the same repository are more similar than those across different repositories.

Inspired by layer similarity, we propose image reconstruction to economize storage and networking. We aim to maximize the overlap between images by regrouping all the files to compose new identical layers. Thus, both storage and networking benefit from enhancing layer sharing. The following questions should be tackled to fully explore the potential of image reconstruction: (1) How many layers are in the reconstructed image? (2) Which layer does a file belong to? (3) What is the order of layers?

Addressing these questions is challenging. First, image reconstruction should keep the validity of images. Docker union mounts each image layer to one directory and only provides a merged view for the container. Altering the layer where the file locates affects the merged result, which may lead the reconstructed image different or unavailable. How to determine and present whether the layer position of files can be altered is nontrivial, which is unsettled in prior work [25]. Second, image reconstruction should trade off the storage usage against the operation overhead. An intuitive idea is to have one file per layer to make each file reusable by all images. However, most of images have thousands of files and about 20% of images have more than 10,000 files [33]. One-file-per-layer can cause the number of layers and the size of metadata to explode. Moreover, too many layers are not conducive to the operation on containers, as a positive correlation between layer depth and operation latency is corroborated in our measurement.

To deal with the above challenges, we first demystify the layered structure and define the commutativity to guarantee the correct order of files and the validity of images. We then propose a mathematical expression revealing the relation between operation latency and image layer depth with a practical data-fitting approach. Correspondingly, the operation cost of images is designed. We formulate the image reconstruction as an integer nonlinear programming (INLP) problem, aimed at minimizing the storage cost and operation cost. To combat the considerable metadata of images, we propose a similarity-aware online image reconstruction algorithm (SOIRA) to solve the INLP. SOIRA rearranges the committed layer to make it be the same as the layer target. Atop the observation that layers within the same repository are more similar, SOIRA selects targets from the repository instead of the whole registry. Thus, the execution time of the algorithm can be significantly reduced. We conduct evaluations with a subset of Docker images, of which the total number of unique files is 1,413,629, and the total size is roughly 60 GB. SOIRA yields about 10% storage saving with only a few more layers compared with state of the art. Additionally, around 9.3% of traffic is saved with each of 20 clients pulling 20 images. It can be inferred that image reconstruction has significant potential for saving both storage and networking in practical large-scale networks.

The main contributions of this paper are listed as follows.

- We explicitly clarify the optimization space of image sharing by assessing the effectiveness of CAS and the redundancy of files. We are the first to quantify and measure the similarity of layers.
- We propose image reconstruction to enhance layer sharing and thus save storage and networking. To our best knowledge, we are the first to define commutativity to guarantee the validity of images in image reconstruction.
- We formulate the image reconstruction as an INLP problem. By exploiting the similarity in layers, we work out SOIRA to combat the massive metadata of images.
- We conduct extensive evaluations driven by images from Docker Hub. Images with 1,413,629 unique files are analyzed to verify the performance of our model and algorithm.

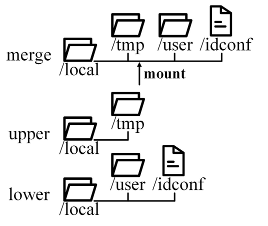
2 BACKGROUND

2.1 Docker Overview

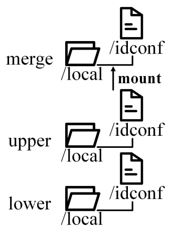
In this part, we introduce the Docker-related background [1] to clarify the layer sharing achieved by CAS and elucidate the origins of similar layers.

Repository and registry. Repository is regarded as the set of different versions of the same type of image. For example, "java:7u121-jdk-alpine" and "java:8u111-jdk-alpine" both belong to the repository "java". Registry is a system for storage and content delivery of docker images. It stores the layers as well as the metadata files (e.g, manifest.json) of images in repositories.

Image delivery. Commands "pull" and "push" are the most two frequently used for image delivery between clients and registries. Docker runs a local daemon at each host. When command "pull" is

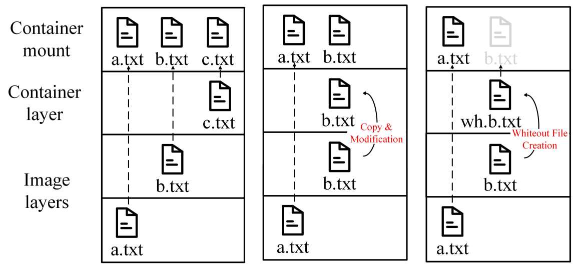


(a) The "user" and "idconf" are merged into the upper layer.



(b) The "idconf" in the upper layer would cover the one from the lower layer.

Figure 2: The diagram of mount.



(a) Creating a new file. (b) Modifying an existing file. (c) Removing an existing file.

Figure 3: The diagram of overlay.

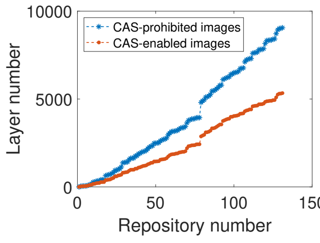


Figure 4: Layer number.

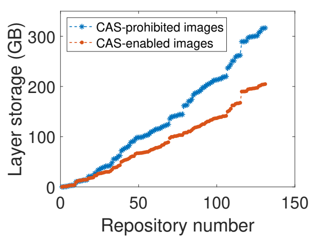


Figure 5: Layer storage.

executed, Docker daemon fetches from the registry the manifest which is a JSON file recording the configuration and the list of layers to compose the image. Each layer is identified and referenced by the digest, which is a content hash with SHA-256 algorithm [21]. Docker daemon checks if a layer is available in local storage based on the layer digest and then solely download layers that are not stored. Consequently, such a scheme known as CAS enables identical layer sharing among different images at one host. Command "push" triggers the Docker daemon to upload new images and then the manifest to the registry.

Image building. There are two methods to create a new image and publish it to the registry: (1) from dockerfile; (2) or through the "Docker commit". Dockerfile is a text file describing the steps for building an image. Each instruction creates a new layer based on the intermediate image generated by previous instructions. "Docker commit" is a command that triggers packaging the running container with its environment to establish a new image. The same and common libraries and packages are referenced in different images. However, developers create and publish their images independently, which possibly invoke libraries with different methods and instructions. As a result, the created layers can be distinct but with lots of identical files.

2.2 Docker Storage Driver

Layer sharing empowered by CAS benefits from the layered structure of images, which is managed by the Docker storage driver. Overlay [3] is a common and promising storage driver employed on Docker. It is based on a union file system named overlayFS [28], which has been written into linux kernel after version 3.18. Union mounting in overlayFS mounts directories to provide a view of single file system for container users. As shown in Figure 2(a), the files and folders in the lower directory are merged into the upper one if they are not exactly in the same path. As shown in Figure 2(b), the files in the upper directory would cover the one with the same path and name in the lower directory, whether the content is identical or not. Copy on Write (CoW) [31] is applied on Overlay. Figure 3 illustrates common operations on the container file system with CoW. Image layers below are read-only, and a writable layer named container layer is initialized with the creation of the container. All the operations are recorded on the container layer. Image layers and the container layer are union mounted to present the container mount view.

3 IMAGE ANALYSIS

In this section, we make empirical studies to motivate our work and provide support for our solution.

Settings. The server is equipped with an eight-core Intel Xeon processor running 2.5GHz with 11GB of ram. We pull 2,200 images being the most recently updated among 130 popular repositories from Docker Hub.

CAS evaluation. Being different from the layer reference count measurement in [33], we compare the layer number and storage of CAS-prohibited images with CAS-enabled images to evaluate the effectiveness of CAS. Figure 4 and Figure 5 show the variation of two metrics versus repositories, respectively. With the increase of repositories, the layer number and storage of images with CAS are always less than without it, and the gap between them gradually gets larger. For all images in our measurement, the storage of CAS-enabled images is 3.25×10^5MB, providing around 38% storage saving.

Image file redundancy. Layer sharing allows to store and transmit unique layers, while data redundancy at file granularity still exists. To reveal the file redundancy, we compute the file number and storage of all unique layers and unique files for each repository. The accumulated results are presented in Figure 6 and Figure 7. For all unique layers in our measurement, the total number of files is 8,305,000, out of which 3,721,000 are unique. Around 55.2% of files are duplicated. The least achievable storage of unique files is 2.1×10^5MB, being less than the storage of unique layers around 35%. *Layer sharing enables partial files reused, while can not eliminate all the file redundancy. There is still a large space for improvement in the storage of images.*

We observe that most of the libraries and packages related with basic operations have formed a fixed layer and serve as a base for other images. For example, application images are often based on

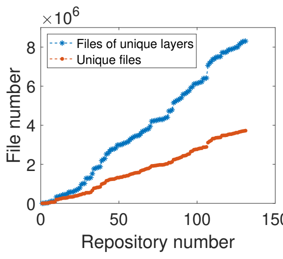


Figure 6: File number.

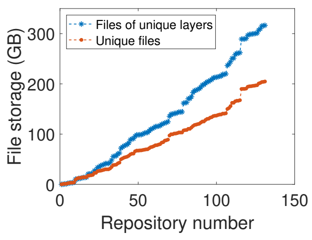


Figure 7: File storage.

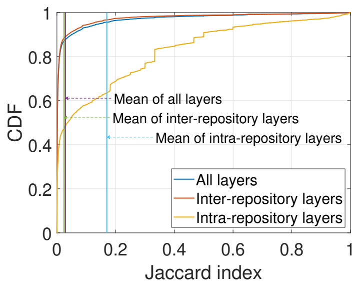


Figure 8: CDF of Jaccard index.

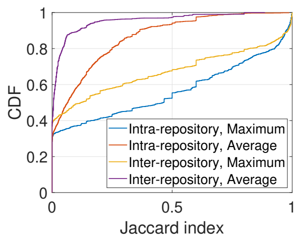


Figure 9: The comparison of Jaccard index.

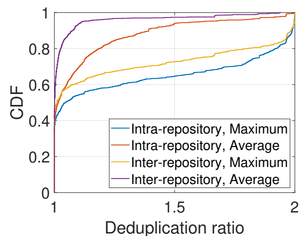


Figure 10: The comparison of deduplication ratio.

operation system images. "ubuntu" is a base image for "ubuntu-upstart", "php-zendserver". "iojs" and "node", both belonging to web framework, share the same five bottom layers. Other libraries and packages that are closely related to the repository type, are also typically called by images within the repository. Nevertheless, these files may belong to different layers since developers do not have uniform specifications for image building. This may be the main reason for file redundancy among images.

Image layer similarity. Substantial identical files lie in different layers, which we regard as similar layers. Now we evaluate the similarity of layers. Two metrics are leveraged to demonstrate the similarity of layers at file granularity. *(1) Jaccard index.* Jaccard index [15] is a common metric for measuring the similarity of two sets. With a range from 0 to 1, the larger value indicates higher similarity. We use it to assess the similarity of layers in terms of file number. We calculate it as the identical file number divided by the total file number of two layers. *(2) Deduplication ratio.* Deduplication ratio [11] is the measurement for data redundancy in terms of storage. It is obtained through dividing the total storage of layer data by the actual storage after deduplication. The value varies from 1 to 2. The larger the ratio, the more storage redundant in the two layers. To the best of our knowledge, this is the first work adopting mathematical metrics to quantify and measure the similarity of image layers.

How many layers are similar? We compute the Jaccard index of each layer pair, and the CDF of results is shown in Figure 8. For the value of all the layer pairs in the image set, about 70% are 0 or close to 0. These layer pairs have several or no identical files. The result of layer pairs across repositories (inter-repository layer pairs) is mostly distributed around 0. For the layer pairs within repositories (intra-repository layer pairs), around 10 percent get the value of Jaccard index more than 0.5, which means more than half of files are the same between each of these layer pairs. Additionally, the average Jaccard index of intra-repository layers is larger than the inter-repository layers as presented. *In summary, not all two layers have identical files. The layer pairs within the same repositories are more similar than those across different repositories.*

How much similar is the layer to layers within repositories and across repositories? With a layer being fixed, we figure out its maximum and average Jaccard index/deduplication ratio with other layers within the repository and across repositories. With all layers fixed one by one, we present the CDF of results in Figure 9 and Figure 10. About 28% of layers get maximum Jaccard index more than 0.8 with other layers in the same repository, while the proportion decreases to 21% when across repositories. The maximum deduplication ratio of around 30% of layers are more than 1.78 with intra-repository layers and more than 1.4 with inter-repository layers. Overall, the curve of maximum/average value of intra-repository layers is below the curve of inter-repository layers.

The Jaccard index difference of a layer is obtained by subtracting its maximum/average value with intra-repository layers from the value with inter-repository layers. We calculate the difference of all layers, and its CDF is shown as Figure 11. Around 68% of the difference of maximum Jaccard index is non-negative, indicating that a higher Jaccard index is achieved within repositories for these layers. In other words, there are 68% of layers can find the most similar layer within its repository. As for the difference of the average Jaccard index, about 75% of layers obtain non-negative value. Figure 12 shows the difference of deduplication ratio between the intra-repository value and inter-repository value. Around 63% of the difference of maximum deduplication ratio is non-negative. Around 80% of layers obtain the average deduplication within repositories no less than across repositories. It demonstrates that the file redundancy in layers within repositories is more serious than layers across repositories. *In general, the layers within the same repositories are more similar and have more redundant storage than layers across the repositories.* This may attribute to the duplicated library files closely related to the repository type. These files are typically called by different versions of images in the repository, while rarely appearing in images of other repositories.

4 SYSTEM MODEL

The problem of image reconstruction is modeled in this section. The set of images is denoted as I. We consider the files with identical content (identified by hash value) and path as the same file, and we denote the set of unique files of all images as K. We denote the set of unique layers by J, and denote the association between existing unique files and layers by $y_{k,j}$. For a newly pushed image i, its file set is indicated as M. In the reconstruction of image i, we have to

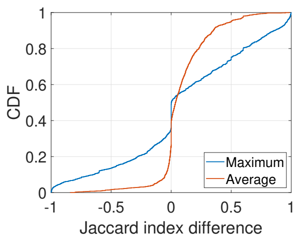


Figure 11: The difference of Jaccard index.

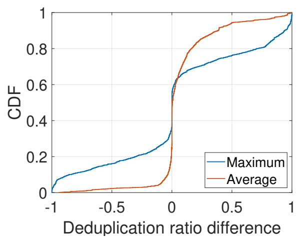


Figure 12: The difference of deduplication ratio.

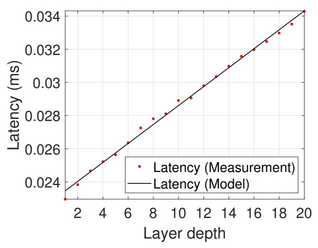


Figure 13: Operation latency with layer depth.

resolve the number of layers denoted as N, as well as the association between files and recreated layers denoted as $\xi_{k,n}$. $\xi_{k,n} = 1$ if and only if file k is in n-th layer, i.e. layer n. It is envisioned to minimize the weighted sum of operation cost and storage cost.

4.1 Operation Cost

It is beneficial to keep images shallow (small number of layers per image) for operation performance reasons. A practical data-fitting approach is conducted to evaluate the operation overhead. We create 30 layers and compose with Overlay. 1000 empty files are contained in each layer. We randomly select 20 files to open with cold cache and measure the latency. The result (an average over 500 runs) is shown in Figure 13. The latency increase with the growth of layer depth. We present the operation latency L as the function of layer depth D, which is denoted as $\mu(\cdot)$. Then we get:

$$L = \mu(D) = a \cdot D + b, \tag{1}$$

where a = 0.001526, b = 0.03087. We define the overall operation cost of an image as the sum operation latency of each layer. With the number of layers of image i being N, the operation cost denoted by C_o^i is calculated as:

$$C_o^i = \sum_{D=1}^{N} \mu(D) = \sum_{D=1}^{N} (a \cdot D + b) = a \cdot \frac{N(N+1)}{2} + b \cdot N. \tag{2}$$

4.2 Storage Cost

We define the storage cost as the incremental storage caused by the unique layers of image i. If the layer n in image i is identical with the layer $j \in J$, then the files contained in them are the same, i.e.,

$$\sum_{k \in M \cup K} |\xi_{k,n} - y_{k,j}| = 0. \tag{3}$$

If there is an existing layer in the registry being identical with layer n, then,

$$\prod_{j \in J} \sum_{k \in M \cup K} |\xi_{k,n} - y_{k,j}| = 0. \tag{4}$$

Otherwise, if layer n is unique to the registry, then,

$$\prod_{j \in J} \sum_{k \in M \cup K} |\xi_{k,n} - y_{k,j}| \geq 1. \tag{5}$$

Denote the size of file k as S_k, then the size of layer n is given as $\sum_{k \in M} (\xi_{k,n} \cdot S_k)$. Therefore, the incremental storage of the new unique layer in image i can be calculated with the formula as follows:

$$C_s^i = \sum_{n \in N} \left\{ \sum_{k \in M} (\xi_{k,n} \cdot S_k) \cdot min(\prod_{j \in J} \sum_{k \in M \cup K} |\xi_{k,n} - y_{k,j}|, 1) \right\}. \tag{6}$$

4.3 Commutativity Model

Due to the hierarchical storage and CoW, there exists order in files. We define the order as the layer difference of files and we employ $\delta^i_{k,k'}$ to denote the order of file k and file k' in the image i. We first let θ^i_k indicate which layer the file k belongs to in the image i before reconstruction. If file k is in the lowest layer, i.e. the first layer, then $\theta^i_k = 1$. If file k is not in image i, then $\theta^i_k = 0$. $\delta^i_{k,k'}$ is calculated with the difference of θ^i_k and $\theta^i_{k'}$ as shown in (7), where $sgn(\cdot)$ is the sign function defined as (8). $\delta^i_{k,k'} = 0$ indicates that file k and file k' are in the same layer. $\delta^i_{k,k'} > 0$ and $\delta^i_{k,k'} < 0$ entail that file k is in the upper or lower layer than file k' respectively. $\delta^i_{k,k'} = -\delta^i_{k',k}$ can be derived. The order of some files can not be changed. Cases are listed as follows:

- If file A in the upper layer is the copy and modification of file A in the same path from the lower layer as shown in Figure 3(b), then the order of them must be kept unchanged in reconstruction.
- If file B is the reliance of file A, then the layer where B belongs to may be lower than the file A. It is envisioned to keep the order of these two files.

We name this relation between files as commutativity. Boolean variable $\lambda^i_{k,k'}$ is leveraged to represent whether file k and file k' are commutative. $\lambda^i_{k,k'} = 0$ means that there is no restriction on the order of two files in image reconstruction, and file k and file k' can be in the same or different layers. $\lambda^i_{k,k'} = 1$ means that the order of two files can not be changed. For the files in the same layer, they must be commutative, i.e., if $\delta^i_{k,k'} = 0$, $\lambda^i_{k,k'} = 0$. For the files in the different layers, the $\lambda^i_{k,k'}$ could be 0 or 1 depending on the file properties. In order to guarantee the validity of images, we must maintain the order of noncommutative files. We denote the order of files after reconstruction by $\delta^{i*}_{k,k'}$, which is given as (9). For files that are commutative, i.e. $\lambda^i_{k,k'} = 0$, the value of $\delta^{i*}_{k,k'}$ can be arbitrary. For two files where $\lambda^i_{k,k'} = 1$, the order of layers must keep consistent before and after the reconstruction. Namely, the value of $\delta^i_{k,k'}$ and $\delta^{i*}_{k,k'}$ must be equivalent. The constraint of $\delta^{i*}_{k,k'}$ for commutativity guaranty is summarized as (10).

$$\delta^i_{k,k'} = sgn(\theta^i_k - \theta^i_{k'}). \tag{7}$$

$$sgn(x) = \begin{cases} 1, x > 0 \\ 0, x = 0 \\ -1, x < 0 \end{cases} \tag{8}$$

$$\delta^{i*}_{k,k'} = sgn(\sum_{n=1}^{N} n\xi_{k,n} - \sum_{n=1}^{N} n\xi_{k',n}). \tag{9}$$

$$\delta^i_{k,k'} \cdot \lambda^i_{k,k'} == \delta^{i*}_{k,k'} \cdot \lambda^i_{k,k'}, \forall k, k' \in K. \tag{10}$$

4.4 Problem Formulation

We formulate the image reconstruction as the problem of weighted cost minimization. It is described as follows:

$$\begin{aligned} \mathbf{P:} \ & \min_{N,\Xi} \ C = \alpha C^i_o + \beta C^i_s. \\ s.t. \ & C1: sgn(\theta^i_k) = \sum_{n=1}^{N} \xi_{k,n}, \forall k \in M \cup K. \\ & C2: \delta^i_{k,k'} \cdot \lambda^i_{k,k'} = sgn(\sum_{n=1}^{N} n\xi_{k,n} - \sum_{n=1}^{N} n\xi_{k',n}) \cdot \lambda^{i_0}_{k,k'}, \\ & \forall k, k' \in M. \end{aligned} \tag{11}$$

$C1$ entails that the files in image i must be consistent before and after reconstruction. $C2$ is obtained by substituting (9) into (10), which guarantees the order of files. $\Xi = \{\xi_{1,1}, \xi_{2,1}, ..., \xi_{M,N}\}$ denotes the decision profile for the file and layer association.

5 IMAGE RECONSTRUCTION ALGORITHM

P is an integer nonlinear programming problem. We make use of the observed similarity between layers and propose the SOIRA to regroup files for the committed images. The similarity between the layer n and the existing layer $j \in J$ is denoted as $T_{n,j}$ and expressed by the Jaccard index, of which the calculation is shown as (12). In the order of the similarity $T_{n,j}$ from high to low, we move files into or out of the layer n to make it be the same as the existing layer j. Since the intra-repository layers have higher similarity than the inter-repository layers overall, only the similarity for layer n with other layers in the same repository is considered, rather than with all layers in the registry. The number of layers in a repository is far less than the number of all layers, which contributes to the reduction of the algorithm execution time. The specific workflow is shown in **Algorithm 1**. When the iteration times exceed the threshold or the layer similarity decreases below the threshold, the reconstruction is terminated.

$$T_{n,j} = \frac{\sum_{k \in M \cap K} (\xi_{k,n} - y_{k,j})}{\sum_{k \in M \cup K} (\xi_{k,n} - y_{k,j})}. \tag{12}$$

Algorithm 1: SOIRA

```
Input: file information of image $i_0$: $\theta^{i_0}_k$, $\delta^{i_0}_{k,k'}$, $\lambda^{i_0}_{k,k'}$
Output: $N$, $\Xi$
1  Initialize $N$ and $\Xi^0$ based on the current image layers and
     files; Set the iteration counter $t$ as 0 ;
2  Compute the current weighted cost denoted as $cost^0$ ;
3  Compute the layer difference $T_{n,j}$ for each layer $n$ ;
4  Choose layers $(n^*, j^*) = arg \max_{n \in M, j \in J} T_{n,j}$ ;
5  Calculate the cost after reconstruction denoted as $cost^{Re}$ ;
6  if $cost^0 < cost^{Re}$ then
7      Keep the image layer unchanged, set $D_{n,j} = -\infty$ and
         goto line 4 ;
8  else
9      Make layer reconstruction, set $cost^0 = cost^{Re}$ and goto
         line 3 ;
10 end
11 $t \leftarrow t + 1$ ;
12 if $(t > t_{threshold})$ or $(\max_{n \in M, j \in J} T_{n,j} \leq T_{threshold})$ then
13     Return N, $\Xi$
14 end
```

6 EVALUATION

6.1 Setup

The algorithm can scale to hundreds of thousands of files in our opinion. However, due to the limitation of network and computation, we only use a subset of images from Docker Hub to make the evaluation. Very-large-scale tests can be done in our future work. For each selected image, layers are addressed to obtain the file set, the size, the position θ^i_k and the order of files $\delta^i_{k,k'}$. The fingerprint (Message-Digest Algorithm [22]) is computed to identify each file in the layer. Since we still have no mature method to accurately determine the commutativity of files just from the limited information, we suppose all the files are noncommutative. We compare our approach with the following benchmark schemes:

- Greedy Offline Image Reconstruction Algorithm (GOIRA). For each file, it iterates all the images that require it, and either picks an existing layer in the image or creates a new one to join in.
- Layered Images from Docker Hub (LIDH). The layers of images are decided by the developers independently.
- One-File-Per-Layer (OFPL). Each file is stored as a layer and then compose the whole image.

6.2 Results

6.2.1 Storage and operation overhead. Figure 14 shows the accumulated storage versus the number of images. Compared with the current layering scheme LIDH, OFPL saves the most significant storage. GOIRA provides storage saving around 1.3% when the image number is 150. The gain of the GOIRA is limited, which attributes to its inherent characteristic. For each unique file, GOIRA iterates all the images that require it and make a decision according to the storage cost and operation cost. The decisions of subsequent images are largely affected by the decisions of previous images. Although the current optimal choice has been made by each image based on the former decision, it may not be the optimal choice as a whole. The proposed approach SOIRA acts as an "inspector", which explores the similarity of layers and adjusts the submitted layer to be the same as the existing layers in the registry. SOIRA yields about 10% storage saving when the image number is 150.

The proportion of storage saving can be further elevated in practice since the number of images in Docker Hub is far more than 150. We notice that the results of SOIRA and OFPL have a large gap. However, this gap is almost impossible to eliminate in consideration of operation overhead. Storing the images with OFPL is superior in terms of storage while behaving extremely terribly in operation performance. Figure 15 shows the accumulated layer number versus the number of images. The subtle storage saving by GOIRA is at the cost of increasing layers. The number of layers with GOIRA increases 7.3% compared with LIDH when the image number is 150. SOIRA only has a few more layers than LIDH. The layer number of OFPL grows rapidly in units of tens of thousands. It reaches 1,413,629 when the image number is 150, far more than the value of LIDH. Additionally, we observe that around 25% of layers are reconstructed, which confirms the feasibility of our approach.

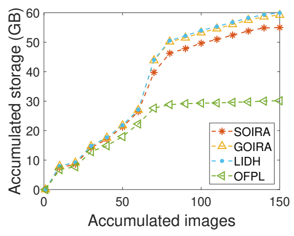


Figure 14: Storage variation of baselines.

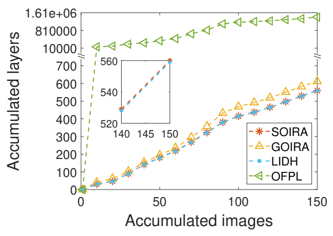


Figure 15: Layer number variation of baselines.

Figure 16 presents the accumulated storage of unique layers versus the number of images for each set of α and β. The ratio of α and β is set to vary from $10:1$ to $1:10$. Only the results with distinguishable differences are presented, and others are omitted. As the increase of $\frac{\alpha}{\beta}$, SOIRA prefers to make image reconstruction to strive for lower storage consumption. With $\frac{\alpha}{\beta}$ being $1:8$, the storage performance of SOIRA is almost the same as LIDH, which entails the images have hardly been altered. When the number of the image is 150, SOIRA achieves around 6.6% storage saving with $\frac{\alpha}{\beta}$ being $1:6$. It enables to save around 8.3% storage when the $\frac{\alpha}{\beta}$ grows to $1:2$. Figure 17 presents the layer number variation for each set of α and β. With a fixed image number, the layer count increase with larger $\frac{\alpha}{\beta}$. When the image number is 150, the sum number of unique layers of LIDH is 559. SOIRA with $\frac{\alpha}{\beta}$ being $4:1$ make the layer count increase around 5.3%. Only a few more layers are added when the $\frac{\alpha}{\beta}$ is $1:4$.

6.2.2 *Network traffic and client storage.*

We argue that image reconstruction enables to save both storage and traffic resources. The metric related to traffic is absent in the optimization cost though, it can be reduced by enhanced layer sharing. Since only layers that are not stored at local would be transmitted, the network traffic of image pulling is the same amount as incremental client storage of images. We suppose 10 clients to randomly download images and the results of network traffic consumption versus the number of images are shown in Figure 18. The network saving of SOIRA compared with GIORA and LIDH increases with the number of images. When the number of images is 50, the network traffic saving is about 8.3% compared with GOIRA and 8.8% compared with

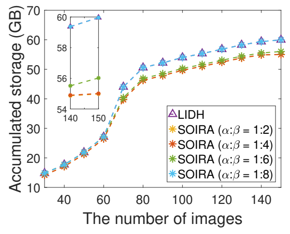


Figure 16: Storage variation of SOIRA.

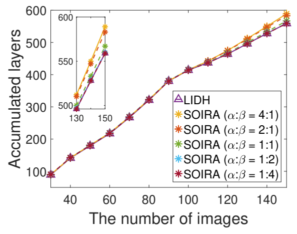


Figure 17: Layer number variation of SOIRA.

LIDH. We then make each client download 20 images and scale the number of clients from 0 to 20. The results of network traffic (client storage) are shown in Figure 19. When the number of clients is 20, the network traffic saving achieved by SOIRA is about 9.3%. The difference of traffic consumption between SOIRA and LIDH increases as the number of images or clients increases, which infers the signification of image reconstruction on saving network traffic in large-scale networks.

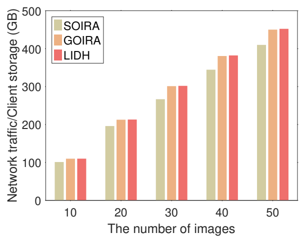


Figure 18: Network traffic/Client storage vs. the number of images.

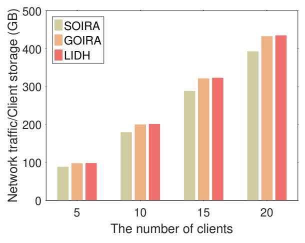


Figure 19: Network traffic/Client storage vs. the number of clients.

6.2.3 *Storage and operation overhead trade-off.*

In the problem formulation, we utilize the weight metrics α and β to trade off the storage cost against the operation cost. With a fixed image set and various sets of weight metrics, we obtain the optimal weighted cost with SIORA and GIORA respectively. The corresponding storage and layer number against different metrics are presented in Figure 20. The value of storage cost and operation cost obviously run in the opposite direction, which indicates that the improvement of one metric may lead to the decline of the another. Less number of layers require more files in each layer, making it hard to enable all files identical in the layer of different images. While more layers in the image result in fewer files of each layer, increasing the probability of layers being the same. The storage of LIDH is 6.4381×10^{10} bytes and the unique layer count is 559. The storage of SOIRA is 5.8772×10^{10} bytes when $\frac{\alpha}{\beta}$ is $1:1$, saving around 9% in storage compared with LIDH. While the unique layer count is 563, only increasing around 0.7%. The curve near the markers approximates the Pareto curve. The Pareto curve of GOIRA is at the upper right of the curve of SOIRA, which means SOIRA achieves less storage consumption than GOIRA with the same layer count. Overall, an improved Pareto frontier is formed and better performance is obtained by SIORA.

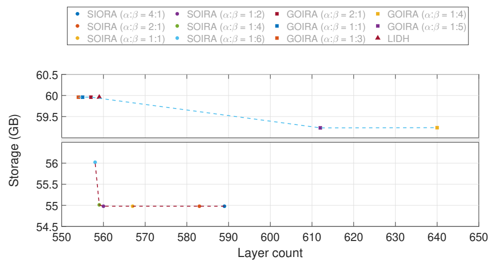
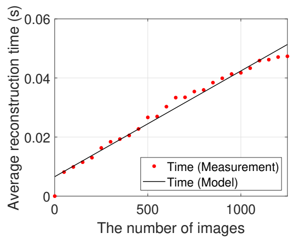


Figure 20: Trade-off between storage and layer count.



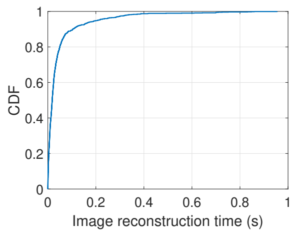


Figure 21: Average image reconstruction time.

Figure 22: The CDF of image reconstruction time.

6.2.4 Image reconstruction time. We conduct image reconstruction with 1250 images and analyze the reconstruction time here. Figure 21 demonstrates the average reconstruction time per image with the increase of images. The average time is 0.012s when 100 images are conducted. The value increases to 0.047s with 1250 images. The image reconstruction time goes through a slight and tolerable increase with the expansion of the image number. Figure 22 shows the distribution of reconstruction time with the total number of images being 1250. Around 80% images have consumed reconstruction time less than 0.05s. It is worth mentioning that our design is easy to scale with more CPU/GPU resources since the reconstruction of each layer is independent.

7 RELATED WORK

Containers are widely researched for design and deployment. Slacker [16] develops lazy fetching where the critical packages for startup are prioritized and launched while others are loaded lazily. Dragonfly employs P2P to speed up image pulling. Firecracker [4] makes prone for containers to accelerate the startup. Checkpoint/restore [10][30] and fork [5][20] are exploited to skip the identical execution procedure. Multi-layer caching [6] is designed for the registry that stores the small layers in memory and large layer in SSD, which speeds up the IO operation and reduces the startup time. For reducing the cold-start executions, a hybrid histogram policy is proposed in [24] to decide the size of pre-warming window and keep-alive window. Large-scale analysis is made in [33] to provide guidance for storage systems. The observations about the Docker images and storage offer the inspiration for our work. All the works mentioned above are prospective while being orthogonal to our work in this paper. The researches focusing on exploring the layered structure and layer sharing are classified into three aspects.

Caching and scheduling. Considering the layer sharing, caching images cooperatively in edge networks is investigated for saving the storage consumption [9]. The fine-planned storage allows more services to be deployed in resource-constrained edge networks, which reduces the retrieval time of images. Microservice placement and request scheduling problem within the layer sharing context is considered in [14][13] to maximize the edge throughput. Joint caching and scheduling strategies enable to save resource consumption on storage and networking. Nevertheless, massive resource wasting by redundancy in layers still exists as only a small proportion of layers are identical and can be shared.

Deduplication for registry. To catch the redundant data in layers, deduplication is an effective method. It enables to save near 23 TB storage occupation from 47 TB unique layer data [33]. Slimmer [34] is proposed to deduplicate registry at file granularity. More complete designing for registry deduplication by them is implemented in DupHunter [32]. Unique files are cached in storage clusters with a metadata database providing the index and recipe mapping files and layers. Pre-fetching and pre-constructing are leveraged by observing user access patterns to mitigate the negative impact of image mapping. Registry deduplication avoids wasting storage resources on duplicated files or data blocks though, it can not reduce the network traffic or client storage footprint.

Image restructure. An empirical study on chunk-based image storage compared with layer-based is introduced in [18]. Random selected 10 versions of images are downloaded to calculate the storage space increment for each version in Docker and Casync (chunk-based image manager), respectively. With variable-size chunking, Casync benefits from the incremental changes and deduplicated files while performing worse than Docker in variation-sensitive cases. Content defined variable sized chunking method enables to determine the identical data blocks and save storage in special cases. While the superiority of layer sharing is abandoned in chunk-based image storage. It is envisioned to reserve the advantages of layer-based images. Researchers from IBM argue to restructure the images, where files are reorganized to compose the layers [25]. A greedy algorithm is leveraged due to the large decision space and evaluations are conducted with the scale of 100 images. However, it is agnostic to the layer reordering, which possibly leads to the alteration and invalidation of images.

8 CONCLUSION

In this paper, we propose a novel image reconstruction model and solution aimed at saving storage and networking. We first make the measurement study to demonstrate the motivation. The effectiveness of layer sharing is evaluated and the file redundancy is revealed. We regard the layers with identical files as similar layers and exploit the similarity with Jaccard index and deduplication ratio. We model the image reconstruction as an INLP problem, of which the storage cost and operation cost are minimized. Driven by the similarity in layers, a similarity-aware online image reconstruction approach is proposed. The experiment results illustrate that our approach achieves better Pareto frontier and overall performance than state of the art.

9 ACKNOWLEDGMENTS

This work was supported in part by National Key R&D Program of China (2020YFB1805502), NSFC (61922017, 61921003, and U21B2016).

REFERENCES

[1] 2021. Docker. https://www.docker.com/.

[2] 2021. Docker Hub. https://hub.docker.com/.

[3] 2021. How the overlay2 driver works. https://docs.docker.com/storage/storagedriver/overlayfs-driver/#how-the-overlay2-driver-works.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th usenix symposium on networked systems design and implementation (nsdi 20)*. 419–434.

[5] 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 *2018 Usenix Annual Technical Conference (USENIX ATC 18)*. 923–935.

[6] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S Warke, Heiko Ludwig, et al. 2018. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 265–278.

[7] Ali Anwar, Lukas Rupprecht, Dimitris Skourtis, and Vasily Tarasov. 2019. Challenges in Storing Docker Images. *login Usenix Mag.* 44, 3 (2019).

[8] Eric A Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing*. 167–167.

[9] Jad Darrous, Thomas Lambert, and Shadi Ibrahim. 2019. On the importance of container image placement for service provisioning in the edge. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–9.

[10] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[11] Mike Dutch. 2008. Understanding data deduplication ratios. In *SNIA Data Management Forum*, Vol. 7.

[12] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. 171–172.

[13] Lin Gu, Deze Zeng, Jie Hu, Hai Jin, Song Guo, and Albert Y Zomaya. 2021. Exploring Layered Container Structure for Cost Efficient Microservice Deployment. *IEEE INFOCOM 2021-IEEE Conference on Computer Communications* (2021), 1–9.

[14] Lin Gu, Deze Zeng, Jie Hu, Bo Li, and Hai Jin. 2021. Layer Aware Microservice Placement and Request Scheduling at the Edge. *IEEE INFOCOM 2021-IEEE Conference on Computer Communications* (2021), 1–9.

[15] Lieve Hamers et al. 1989. Similarity measures in scientometric research: The Jaccard index versus Salton's cosine formula. *Information Processing and Management* 25, 3 (1989), 315–18.

[16] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 181–195.

[17] Devki Nandan Jha, Michael Nee, Zhenyu Wen, Albert Zomaya, and Rajiv Ranjan. 2019. SmartDBO: smart docker benchmarking orchestrator for web-application. In *The World Wide Web Conference*. 3555–3559.

[18] Yan Li, Bo An, Junming Ma, and Donggang Cao. 2019. Comparison between Chunk-Based and Layer-Based Container Image Storage Approaches: an Empirical Study. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 197–1975.

[19] Paul B Menage. 2007. Adding generic process containers to the linux kernel. In *Proceedings of the Linux symposium*, Vol. 2. 45–57.

[20] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 57–70.

[21] Dian Rachmawati, JT Tarigan, and ABC Ginting. 2018. A comparative study of Message Digest 5 (MD5) and SHA256 algorithm. In *Journal of Physics: Conference Series*, Vol. 978. IOP Publishing, 012116.

[22] Ronald Rivest and S Dusse. 1992. The MD5 message-digest algorithm.

[23] Mendel Rosenblum and Tal Garfinkel. 2005. Virtual machine monitors: Current technology and future trends. *Computer* 38, 5 (2005), 39–47.

[24] 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 *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218.

[25] Dimitris Skourtis, Lukas Rupprecht, Vasily Tarasov, and Nimrod Megiddo. 2019. Carving perfect layers out of docker images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[26] Piet Smet, Bart Dhoedt, and Pieter Simoens. 2018. Docker layer placement for on-demand provisioning of services on edge clouds. *IEEE Transactions on Network and Service Management* 15, 3 (2018), 1161–1174.

[27] Gaetano Somma, Constantine Ayimba, Paolo Casari, Simon Pietro Romano, and Vincenzo Mancuso. 2020. When Less is More: Core-Restricted Container Provisioning for Serverless Computing. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 1153–1159.

[28] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. 2017. In search of the ideal storage configuration for Docker containers. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 199–206.

[29] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Thomas C Bressoud, and Adrian Perrig. 2003. Opportunistic Use of Content Addressable Storage for Distributed File Systems.. In *USENIX Annual Technical Conference, General Track*, Vol. 3. 127–140.

[30] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[31] Frank Zhao, Kevin Xu, and Randy Shain. 2016. Improving copy-on-write performance in container storage drivers. In *Storage Developers Conference*.

[32] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R Butt. 2020. Duphunter: Flexible high-performance deduplication for docker registries. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 769–783.

[33] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Arnab K Paul, Keren Chen, and Ali R Butt. 2020. Large-Scale Analysis of Docker Images and Performance Implications for Container Storage Systems. *IEEE Transactions on Parallel and Distributed Systems* 32, 4 (2020), 918–930.

[34] Nannan Zhao, Vasily Tarasov, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit Warke, Mohamed Mohamed, and Ali Butt. 2019. Slimmer: Weight loss secrets for docker registries. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 517–519.