Multi-grained Trace Collection, Analysis, and Management of Diverse Container Images

Zhuo Huang, Qi Zhang, Hao Fan *Member, IEEE*, Song Wu *Member, IEEE*, Chen Yu *Member, IEEE*, Hai Jin *Fellow, IEEE*, Jun Deng, Jing Gu, Zhimin Tang

Abstract—Container technology is getting popular in cloud environments due to its lightweight feature and convenient deployment. The container registry plays a critical role in container-based clouds, as many container startups involve downloading layer-structured container images from a container registry. However, the container registry is struggling to efficiently manage images (i.e., transfer and store) with the emergence of diverse services and new image formats. The reason is that the container registry manages images uniformly at layer granularity. On the one hand, such uniform layer-level management probably cannot fit the various requirements of different kinds of containerized services well. On the other hand, new image formats organizing data in blocks or files cannot benefit from such uniform layer-level image management. In this paper, we perform the first analysis of image traces at multiple granularities (i.e., image-, layer-, and file-level) for various services and provide an in-depth comparison of different image formats. The traces are collected from a production-level container registry, amounting to 24 million requests and involving more than 184 TB of transferred data. We provide a number of valuable insights, including request patterns of services, file-level access patterns, and bottlenecks associated with different image formats. Based on these insights, we also propose two optimizations to improve image transfer and application deployment.

Index Terms—Container image, On-demand image, Cache, Prefetch.

1 Introduction

LIGHTWEIGHT containers have emerged as a crucial technology for cloud platforms. Images serve as the delivery mechanism for containerized services and are integral to the entire life-cycle of these services. As shown in Fig. 1, developers release their services by building a layered image that includes system files, packages, and service contents. Once built, the image is uploaded to a container registry. Users can easily download the image from a container registry and deploy a container from the image on servers, regardless of the underlying architecture and environment.

Efficient image management, including transfer and storage, is essential for establishing a container cloud platform with rapid response and efficient operation and maintenance [1]. However, for the traditional image format proposed by Docker [2], the complete image needs to be downloaded when deploying a container, which can account for up to 76% of the container deployment time [3]. To accelerate image transfer, researchers propose a series of optimization strategies, such as cache [4], prefetch [5], [6], and *Peer-to-Peer* (P2P) [7], [8]. They also propose novel on-demand image formats [3], [9], [10], [11], which only download a small portion of data on demand when starting a container, enabling fast deployment.

Nevertheless, with the dramatic development of

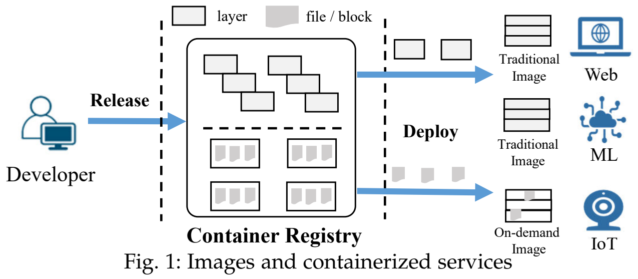


Fig. 1: Images and containerized services

container-based platforms, new problems arise in managing images within a container registry, even if the mentioned optimizations are applied [5], [6], [12], [13]. Firstly, managing image data uniformly at layer granularity is not enough [14], [15], [16]. Both image- and file-level information are desired to enable further optimization. For example, file-level prefetching is required to optimize on-demand images. Secondly, different kinds of containerized services need their corresponding optimal image formats. The performance of on-demand image format is not always better than that of traditional image format [3], [6], [9]. For example, on-demand images need to be converted from traditional images in a container registry, which makes on-demand images unavailable before the conversion finishes [9]. Our container-based cloud platform covers a diverse set of users with various services. These services are developed in different paradigms, maintained in different deployment modes, and deployed in nodes with different performances. We need to understand how these factors affect image data transfer and existing acceleration solutions.

Accordingly, we perform comprehensive analysis on the data transfer behaviors of a variety of deployed services with different request granularities and image formats, including analytics, *Internet of Things* (IoT), *Machine*

- *Z. Huang, Q. Zhang, H. Fan, S. Wu, C. Yu, and H. Jin are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China. H. Fan, S. Wu, and H. Jin are also with Jinyinhu Laboratory, Wuhan, China. Hao Fan is the corresponding author. Email: haofan@hust.edu.cn.*
- *J. Deng, J. Gu, and Z. Tang are with Alibaba group.*

Learning (ML), web, and platform workloads. These have different computing paradigms (e.g., monolithic, or decoupling), development modes (e.g., traditional mode, or DevOps mode), and node performances (e.g., network, or storage space). We design a toolkit, named *mTracer* to capture operational traces in three granularities: image-level, layer-level, and file-level. Image traces are collected from a container registry of one of the leading cloud service providers. The traces cover 30 days, total over 24 million requests, and involve over 184 TB of data transferred. We have open sourced *mTracer* at: `git@github.com:CGCL-codes/Multi-grained.git`, and the traces will be available for public use upon obtaining the authorization to facilitate further research.

We obtain several insightful observations. For example, image traces of different services exhibit different characteristics (e.g., image size, pull interval) due to the changes in computing paradigms, development modes, and node performance. For example, it inspires us to design a lifetime-based eviction strategy for the container registry cache based on the observation that over 70% of images stay active in less than 3 hours for web services due to decoupling and DevOps, and most images of other services usually stay active for more than 11 days. In terms of image-level requests, we find that many images tend to be pulled in pairs even if they are on different servers from the view of the container registry. This motivates us to design an image-level prefetching mechanism. On-demand images, which are replacing traditional images, are not always the best choice. The on-demand images are converted from traditional images. Pull requests directly after uploads, which are quite common (over 80% of the first pull is within 1 second after an upload), cannot be served during the conversion.

We present two optimizations to clarify the practicality of our observations. First, *Lifetime and Association-based Cache* (LACache), is a layer-level cache strategy in the container registry. LACache evicts layers based on services and prefetches images based on their associations. LACache can reduce 57% layer provisioning latency for the container registry compared to the state-of-the-art works. Second, IFRecommender, employs a neural network to recommend image formats for services based on request interval, conversion overhead, and bandwidth. IFRecommender can reduce 42% the deployment latency of containers compared to the situation when all services use on-demand images.

2 Background and Motivation

In this section, we introduce the structure and management of images and explain the motivation for our analyses.

2.1 Organization and Storage of Images

Currently, the most widely deployed image is proposed by Docker and promoted by *Open Container Initiative* (OCI) [17]. The traditional image is designed as a layered structure which brings two benefits. The first is the convenient building. For a container, a developer creates a directory, named layer, to hold data generated by each step and commits to confirm the step. The developer can build a new layer upon any of the earlier committed layers. The second is the data sharing. For two layers, if their direct lower layers and their own fingerprints (e.g., SHA256) are identical separately, they can be shared.

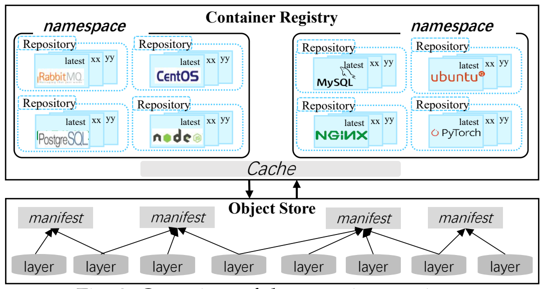


Fig. 2: Overview of the container registry

Container registries, such as Docker Hub [1] and IBM Cloud Container Registry [5], acts as a storage and content delivery system for container images. As shown in Figure 2, in our container registry, images of various services are put into different namespaces. In each namespace, users can create repositories, which hold images for a particular service. Images can have different versions, called tags. For an image, the container registry stores two parts: compressed tarballs of layers and manifests that record which layers make up the image. Layers of different images are stored together in the object storage for sharing.

2.2 Data Transfer of Images

For traditional images, an intact image must be pulled before starting a container. Specifically, the client sends a "`PULL()`" command to the local daemon. The daemon then requests the manifest of the image by issuing a "`GET()`" request to the container registry. Based on the manifest, the daemon downloads layers that are not present in the client. An absent layer can be searched and pulled by issuing a "`HEAD()`" request and "`GET()`" request to the container registry, respectively. The container registry can redirect layer requests to a different URL, e.g. to an object store, which stores the actual layer. The client can upload an image by sending "`PUSH()`" to the local daemon. In contrast to "`PULL()`", "`PUSH()`" works in reverse order.

On-demand Image. As not all image data are accessed when running a container [3], researchers design on-demand images [2], [9], [10], [18] that only download necessary files or blocks during the container's run. Traditional images do not support on-demand downloads, as the layers are compressed in the container registry and cannot be accessed randomly. Therefore, data in on-demand images are reorganized. For example, DADI [2] organizes image data in the granularity of blocks and locates blocks with a virtual block device. Slacker [3] organizes and locates image files through network file systems. Note that as most clients cannot directly build on-demand images, on-demand images are converted from traditional images in the container registry, which costs extra time and resources.

2.3 Image Characterizations

With the diversification of services and the emergence of new image formats, the request patterns of images require further characterization for efficient image management.

Characteristics of Different Services. Due to differences in computing paradigms, development modes, and node performance, images of services exhibit different request patterns. However, existing analyses are for the general features of the whole container registry, lacking the characterizations on different categories of services [5], [12]. For example, most images of traditional analytic services are widely distributed, which means P2P is suitable. In contrast, only a fraction of images on serverless web services are frequently pulled. Therefore, we need to analyze images based on the computing paradigm, development mode, and node performance of services.

Request Characteristics in Different Granularities. As images are pulled in the granularity of layer, existing analyses mainly character request patterns at the layer level [6], [14], [15]. On the one hand, a complete image is required when deploying a container based on the traditional image. The image-level analysis is required to observe the impact of image pulling on deployment and reveal relationships between images. On the other hand, on-demand images manage data at the granularity of blocks or files. File-level pattern analysis is required to optimize on-demand images.

Comparison of Different Image Formats. The performances of traditional and on-demand images vary depending on image size, data usage, and network. For example, when images are tiny, which is common in platform, deploying containers based on traditional images is sometimes faster than on-demand images. However, cloud users often replace on-demand images with traditional images regardless of the service. This delays the deployment of some services and even causes some of the users to roll back to traditional images. A comprehensive comparison between traditional images and on-demand images is required to guide the users in choosing appropriate image formats.

3 Methodology

In this section, we introduce selected services and explain how to obtain image-level and file-level traces.

3.1 Traces of Different Categories of Services

To facilitate a comprehensive analysis of container image usage, we collect a huge number of image traces from a cloud provider. The deployment of containerized applications is affected by factors such as image size, bandwidth, and update frequency. These factors are impacted by the degree of decoupling, deployment location, and development mode. Therefore, we have selected services with varying degrees of decoupling, deployment locations, and development modes for observation. Additionally, these services represent the most commonly used ones in our business.

For high scalability and high resource utilization, an important trend of cloud containerized applications is to be decoupled into some inter-linked execution units [19]. We classify the container-based services into three categories according to the degree of decoupling, i.e., monolithic services, lightly decoupled services, and highly decoupled services, as shown in Table 1. For monolithic applications, a category of services in the data center and a category of services outside the data center are selected. The decoupled application is mainly deployed in the data center. A lightly decoupled service simply decouples the data from the functions, while a highly decoupled service separates the functions. For

TABLE 1: The description of different categories of services

	Level of Decoupling	Representative Services
Cloud Service	monolitic	analytics
		IoT
	lightly decoupled	ML
	highly decoupled	web
		platform

decoupled applications, a new development mode, DevOps mode[1], is becoming increasingly popular. We select web services and platform services that utilize DevOps and traditional development modes for observation, respectively. It is noted that the services we chose also appear in the typical cases of leading cloud service providers [20], [21], [22]. The details of each category are as follows:

Monolithic Services. This kind of service is usually containerized in a straightforward way. All required data of an application are put in a single image.

The analytics services are the first to be deployed in container cloud platforms, like MapReduce [15]. Due to a lack of practical experience with containers, these containerized services are monolithic. The use of only one image for a service inevitably results in a large image.The key feature of IoT services is that their resources are physically limited [23], and thus images for IoT are tailored and simplified.

Lightly Decoupled Services. ML services are newly emerged on the cloud for both research and engineering. ML services are roughly decoupled into datasets and models sometimes. Due to the large size of data involved (e.g., oversized models and training datasets), images of ML probably exhibit different features.

Highly Decoupled Services. This kind of service is deployed with full consideration of characteristics of container cloud platforms (e.g., easy scaling and high resource utilization), meaning that a service is decoupled into numerous containerized execution units. Combining with real-world scenarios, we select two representative services.

Serverless computing is an increasingly popular computing paradigm. Web services are pioneer and typical services migrated to serverless platforms [19]. These services are decoupled into numerous fine-grained handling units, called functions, to make it easy to scale rapidly. With the use of DevOps mode, images are updated frequently.

Apart from services that are highly decoupled by users, the cloud provider will help users accomplish the deployment of basic environments, such as kubernetes and prometheus. Such services are also highly decoupled into multiple components and account for a high percentage of image pulling because of their wide use.

The traces are derived from the logs of the container registry. All the traces have been anonymized and our trace-based analysis is conducted in a secure environment with the authorization of the trace provider. Table 2 shows the basic characteristics of each collected category of services. Note that the nodes of ML and IoT are fewer than other categories, because there are proxy nodes that will store cache images and only the uncached images will be pulled from the container registry. Proxy nodes also results in a decreased frequency of pulls and distribution of nodes for layers. As we can only collect requests from the container

1. **DevOps mode**: Developers and operations work in close collaboration to enable rapid releases of new software features.

TABLE 2: The description of the real-world traces

Category	Request Number	Image Number	Node Number[1]	Image Pulled Size (GB)	Image Average Size (MB)	RPG[2]
Analytics	1,200,000	20	5,000	100,000	4,150	1:1,122
IoT	66,000	300	100	20	100	1:123
ML	330,000	300	500	50	300	1:76
Web	2,700,000	34,000	42,000	5,000	500	1:20
Platform	19,900,000	5,200	250,000	78,500	14	1:7,746
Total	24,196,000	39,820	297,600	183,570	434	-

[1] The node refers to the request source which can be a virtual machine, a proxy, or a switch, etc.
[2] *RPG* means that the Ratio of PUTs to GET.

registry, we only observe the pull behavior directly related to the container registry. Each request of the traces involves multiple fields as shown in Table 3. The traces cover 30 days in the third quarter of 2021. The total amount of traces is over 12 GB including over 24 million requests, involving over 39 thousand images whose size exceeds 183 TB. These requests are from about 298 thousand nodes. And we also classify all requests that are related to the pushing and pulling operations of images. To promote future research, all traces will be open-sourced after the authorization progresses.

3.2 Deriving Multi-grained Traces

The layer-level traces are collected by capturing HTTP requests from the container registry. As layer-level traces are collected from the container registry, the details of the user are lacking, and the pull served by cache outside the container registry cannot be collected. There are some challenges in performing image- and file-level analysis: (1) the container registry can only capture layer-level traces. As layers can be shared by images, layer requests do not have a label recording to which one image they belong. This prevents the container registry from directly detecting which layer requests belong to the same image while serving a large number of layer requests; (2) in the view of the host, retrieving detailed information of file requests from a container is challenging. On the one hand, requests from containers are mixed with regular process requests in the host. We need to distinguish container requests from all host requests. On the other hand, the information carried in the request only contains the logical view path of the file in the container, which lacks knowledge of the image layer. We still need to find which layers the files belong to.

TABLE 3: The collected fields in image traces

Tag	Description
Image: namespace	name of the namespace
Image: repo	name of the repository
Image: tag	image version
Layer: digest_id	digest id of the layer
Layer: size	size of the object
Request: method	HTTP method
Request: timestamp	the time received by the server
Request: ip	IP address
Request: user_agent	header of the HTTP request
Request: api	the request type
Request: network_type	network type
Response: body_length	size of the HTTP response body
Response: response_time	HTTP response time
I/O: timestamp[1]	timestamp of the the I/O operation
I/O: container_id[1]	container id of the I/O operation
I/O: inode[1]	inode number of the accessed file
I/O: data_size[1]	size of I/O data
I/O: image_layer[1]	the accessed image layer

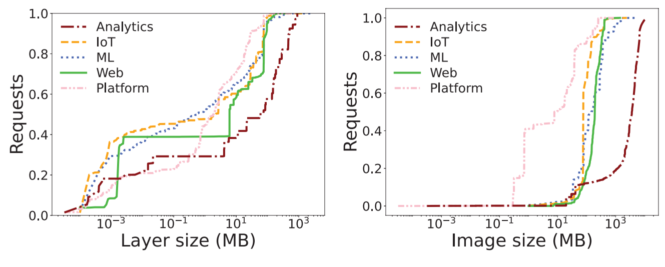
[1] Tag "I/O"s are collected from the client side and others are from the container registry.

TABLE 4: The comparison of container image analysis tools

Tool	Image Analysis			Data Access	
	layer level	image level	file level	startup I/O	runtime I/O
Duphunter[7]	✔				
Crawler[8]	✔				
Hellobench[10]	✔			✔	
DADI[25]	✔		✔	✔	
Tracee[4]			✔		
Replayer[15]	✔				
mTracer	✔	✔	✔	✔	✔

Image-level Analysis. We propose an aggregation approach that runs offline based on the layer-level traces. Specifically, we regard requests with time intervals less than a specified threshold (empirically set to 5 minutes) and with the same address (i.e., the IP, port number, and repository name of the request) as requests for different layers of the same image. An exception only occurs when requests with the same IP and port pull more than one image in the same repository within a time interval. In response to this, we will further rectify the aggregated image information by reviewing the subsequent pull traces. We compare the exact log information obtained from the clients in our business with the results aggregated in the container registry. The result shows that all the tested logs (i.e., over 20,000 logs) are aggregated correctly. Because the aggregation is an offline operation, the overhead does not affect system operations. Given the importance of image-level information and the complexity of recovering it, we also submit a proposal to the official container community to add a related field to the future version of the container image specification.

File-level Analysis. First, *eBPF* technique [24] is used to capture I/O requests in virtual file system and filters container I/O requests based on cgroup id. As listed in Table 3, the fields collected from an I/O request are `timestamp`, `container_id`, `inode`, and `data_size`. The traces are merged in memory and written to storage when the I/O burden is low. Then, the logical path is switched to the physical path by parsing the inode information offline to minimize overhead. Finally, we find layer information in the physical path and add it to each trace item. In particular, the size of the collected data is 27MB while reading 1 GB of image data at the granularity of 4KB. In this paper, the file-level traces of an image are obtained by deploying containers on a standalone server. Due to the stateless nature of container images, image file access is constant when the same containers are deployed. We deploy *xwiki* which involves accessing about 200MB of image data and evaluating the overhead introduced by *mTracer*. *mTracer* increases CPU overhead by about 4% and memory usage by about 70MB. Most of the overhead comes from running *eBPF* and recording traces. As we use data aggregation and regular write-back to reduce overhead, the image deployment time varies by less than 5% (ranging from 39s to 41s). We integrate the above solutions into a multi-grained tracing toolkit, named *mTracer*. Table 4 compares *mTracer* with existing tracing tools.



(a) Layer (b) Image

Fig. 3: CDF of pull requests for layer size and image size

4 Real-world Workload Analysis

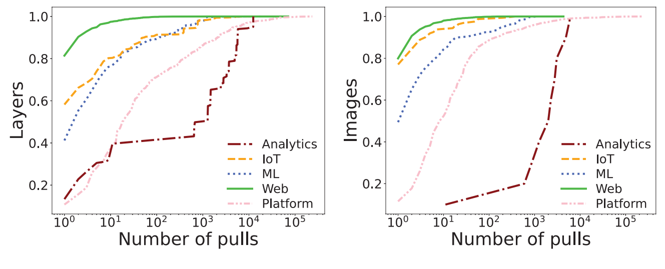
In order to provide bases for possible optimization methods, such as caching and prefetching, we analyze the pull requests and file access of images:

1. We analyze the characteristics of pull requests for various services, including size, time interval, and association, because image pulls directly affect the deployment efficiency of containers.
2. We analyze the file access patterns of container deployment, including the type of files, redundancy, and distribution across layers.
3. We compare the performance of traditional images with that of on-demand images, evaluating container startup, service startup, container run, and conversion time.

4.1 Pull Analysis of Different Services

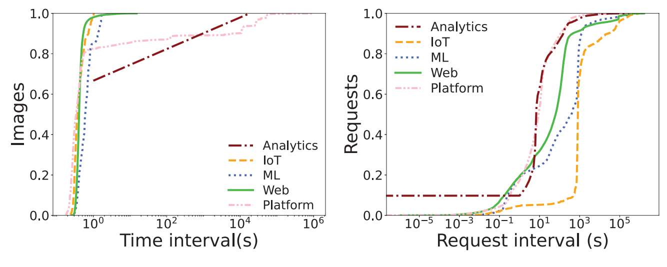
Sizes of Pulls. Fig. 3(a) shows the *cumulative distribution functions* (CDF) of pull request for layer size. It reveals similar trends across all kinds of services. Specifically, approximately 40% of pulls are for layers that are smaller than 10 MB, while about 20% are for layers that exceed 100 MB in size. Pulling large layers is because deploying containers often involves using several base layers that contain the dependencies and environments. Existing work designs cache at the granularity of layer and preferentially cache small layers, because most layers are small and suitable for caching [5]. However, as we observed, a substantial number of pull requests are for large layers. This means that some large layers should also be preferentially and cautiously cached as they are frequently pulled and may evict many popular small layers.

Starting a service with a traditional image requires the entire image to be downloaded, which means that image size determines the deployment efficiency. As shown in Fig. 3(b), we observe the size of image data downloaded when deploying services. For analytics services, over 80% of image pulls are larger than 1GB, while for IoT services, web services, and ML services, more than 80% of image pulls are between 100MB and 400MB in size. As cloud technology advances, a service is decoupled into micro-services or functions, and the content of the image is simplified, making the image smaller. However, currently, transferring images larger than 100MB is still time-consuming in bandwidth-limited situations, highlighting the need for image-level caching and prefetching.



(a) Layer (b) Image

Fig. 4: CDF of layers and images for the number of pulls



(a) Interval of upload and pull (b) Pull interval

Fig. 5: CDF of image and request for request intervals

Number of Pulls for an Image. Fig. 4 shows the CDF of layer and image for the number of pulls, highlighting the variation in the number of pulls. For services in analytics, about 80% of the images are pulled over 400 times, indicating that these images are used frequently. Images of analytics services, which employ traditional paradigms, are stable and often used for a long time. In contrast, for web, IoT, and ML services, only about 20% of the layers or images are pulled over 100 times. Especially for web services, only 5% of the layers or images are pulled over 100 times. Emerging services tend to be decoupled or simplified. When deploying a service, only small images containing desired micro-services or functions need to be pulled, making pulls highly centralized. This means caching mechanisms can effectively accelerate image transfer for emerging services.

Time Intervals of Pulls. We depict the time interval from upload to the first pull of an image as shown in the Fig. 5(a). We find that images are generally quickly pulled after being uploaded. For example, for images of web services, over 90% of images are pulled within 1 second after upload. We also observe the average pull interval of images. As shown in Fig. 5(b), over 50% of images in ML, web, platform, and analytics services are over 10 seconds, and 95% of images in IoT services are over 1000 seconds. This means that the time interval between subsequent pulls of the image becomes longer. Notably, immediate pulls right after uploads for on-demand images may experience delays, because on-demand images are not available until they are converted from traditional images. Additionally, as shown in Table 2, the impact of this conversion delay may be more pronounced for emerging services that update quickly due to DevOps, such as web services.

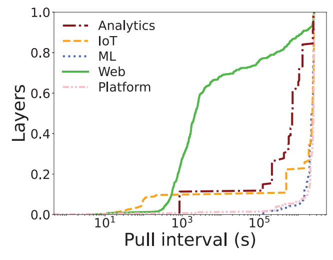
Active Duration of a Layer or an Image. Fig. 6 shows time intervals between the first pull and the last pull request of the same image. For analytics, IoT, platform, and ML services, most images and layers are active for a long time (over 1 day). However, for serverless, more than 40% of images receive requests within 16 minutes. This indicates that the versions of these images are unstable and will be updated iteratively. Serverless decouple its services and employs DevOps mode, which leads to its images being frequently pulled but with a rapid decline in popularity.

This results in the serverless images changing quickly, and layers of serverless in the cache can be quickly evicted.

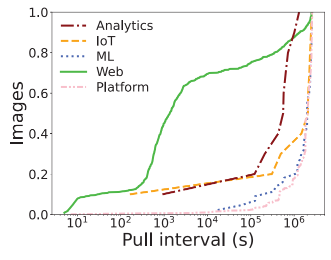
Distribution of Pulls. Fig. 7 shows how many nodes will pull a certain image or layer. First, analytics services that are developed using the traditional development mode are stable. Second, a analytics service is monolithic, and every node that runs the service needs to pull the entire image. Therefore, images of traditional services tend to be distributed widely (i.e., over 30% of images are deployed in 1000 nodes), which makes P2P acceleration suitable for their deployment. In contrast, only a few images from emerging services are widely distributed (i.e., about 50% of the images are pulled from less than 10 nodes). After services are tailored or decoupled, each node downloads only require small images. As a result, only a small fraction of stable images are widely distributed and suitable for P2P acceleration.

To confirm that images deployed on only a few nodes are not suitable for P2P acceleration, we perform a simulation experiment with *NS3* [25]. We simulate a cluster of 100 nodes, randomly connected. The latency between adjacent nodes is 20ms, and the latency from nodes to the container registry is 150ms. As shown in Table 5, the time it takes for a node to pull a 10MB layer at a bandwidth of 10MB/s. When the layer is only distributed in 1% of nodes, the pull time is 2.38 seconds, due to the search procedure. Pulling directly from the container registry takes 1.49 seconds.

Associations between Different Images. We further observe the association between the pulls of different images and find that some images tend to appear in pairs. For an image (*image A*) that is pulled more than 100 times, if another image (*image B*) is pulled soon (within 60s) after more than α% (empirically set to 50) of *image A*'s pulls, we consider the two images are associated, that is, an image pair. However, image pairs are not always pulled together, as one of the images may have been pulled to the client while running other services. Table 6 shows the number of image pairs in each category. We find 412 image pairs from 252 repositories. Because several components should run in collaboration with each other. For example, *kubelet* and *proxy* are usually deployed together. We also find 286 image pairs from 167 repositories among web services, which is likely because services are usually decoupled into several functions. For example, *nginx* and *zuul* should be deployed to run a *Spring* service. Additionally, ML services often separate models and datasets, generating some image pairs.



(a) Layer

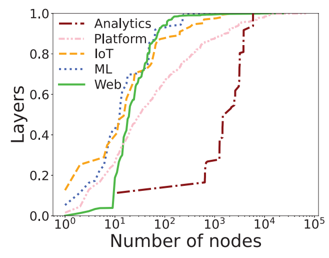


(b) Image

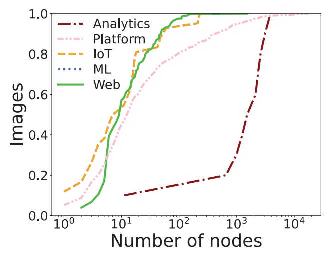
Fig. 6: CDF of layers and images for time intervals between the first pull and the last pull

TABLE 5: Comparison of existing storage systems

Distribution	P2P 1%	P2P 10%	P2P 20%	P2P 30%	Registry
Time(s)	2.58	1.83	1.68	1.55	1.69



(a) Layer



(b) Image

Fig. 7: CDF of layers and images for the number of nodes that a certain image or layer is deployed

TABLE 6: Comparison of existing workloads

	Web	Platform	ML	IoT	Analytics
Repositories	>160	>250	>380	>50	>5
Pairs	286	412	50	0	0

Note that detecting image pairs on the client side is difficult, as image pairs sometimes are deployed on different nodes.

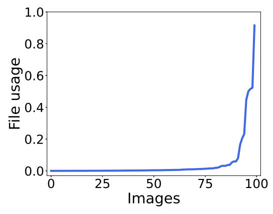
A service tends to be decoupled into several micro-services or functions to enable fine-grained scheduling and billing. Unfortunately, this may delay the deployment of a service, because the container registry serves requests at the layer level, and the images of micro-services or functions are pulled one after another causing multiple network accesses. By letting the container registry be aware of the image associations, the associated images can be provided in advance, thereby accelerating service deployment.

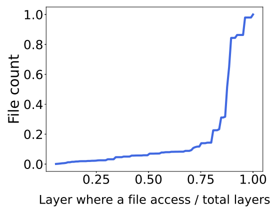
4.2 File-level Access during Deployment

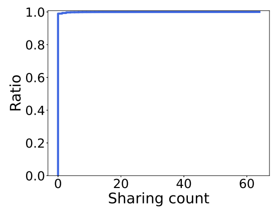
We further observe image file access (e.g., usage, type, location, and duplication) of container deployment to guide the optimization of on-demand images. We select the top-100 downloaded images[2] and replay their latest version locally to observe file access.

Fig. 8(a) shows the usage of image files until the service is ready. On average, about 4.3% of the image files are accessed to start a service, and 90% of the images only access less than 10% of files. We also categorize the accessed files of all the images. Fig. 8(b) shows the layers in which the accessed files are located. It is observed that most of the accessed files are located in the top layers of images. We can accelerate on-demand images by prefetching files in top layers of the image. Fig. 8(c) shows CDF of all accessed files for sharing count. Almost all of these files are unique (only about 1% of the files are redundant). This means that a file-level or block-level cache on the client cannot accelerate the access of on-demand images when different kinds of services are deployed on a single client. Furthermore, duplicated accesses occur mainly within the same repository. Fig. 8(d) shows the redundancy of accessed files within each repository (two versions of images in each repository). The redundancy of accessed files between two versions of the same service reaches 70%. This implies that different versions of the same service exhibit similar I/O patterns. It is easy to accelerate the deployment of on-demand images

2. Top-100 downloaded images: We are not authorized to carry out file-level analysis on images from production environments that need to invade into these images, so we get the names of the top-100 downloaded repositories from the cloud provider and get their corresponding latest images from the public registry. File-level analysis and image format comparison are based on these images.


(a) Data amount


(b) Distribution


(c) File sharing

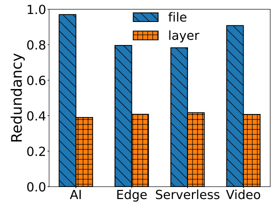
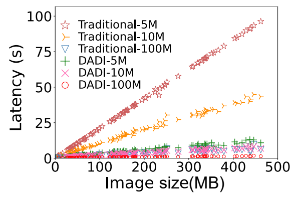

(d) File redundancy

Fig. 8: Access pattern of files. The ratios are calculated based on file counts.


(a) Container startup latency

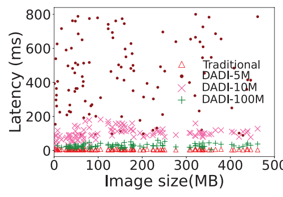

(b) Service response latency

Fig. 9: The latency of container startup and service response. The x-axis represents that the sizes of traditional images sorted in ascending order. The y-axis shows the latency of container startup or service response.

by recording file access of a repository and prefetching files for clients that pull other images from the repository.

4.3 Traditional Image v.s. On-demand Image

We select the top 100 downloaded images to compare the performance of traditional images and on-demand images. The working of images can be into four phases, i.e., container startup, service response, container running, and image conversion. For the on-demand image format, we choose DADI [2] as the evaluated subject. DADI is an open-sourced project promoted by container community [17], which is increasingly popular on commercial platforms.

Container Startup. This phase starts when a pull request is sent and ends with the container creation. We evaluate the container startup of two image formats under different network bandwidths. As shown in Fig. 9(a), when the network bandwidth is 5 MB/s, the latency of traditional images is 42 seconds for the image with a size of 200 MB, which is 9× higher than that of the on-demand image. When the bandwidth is 100 MB/s, the gap in startup latency between these two image formats is slight. Thus, as the bandwidth in cloud platforms gets larger, the advantage introduced by on-demand images is gradually decreasing. When the network bandwidth is limited (e.g., IoT), using the on-demand images can significantly improve deployment.

Service Response. This phase is from the time when service inside the container boots up to the time when user requests are responded to. During this phase, the data involving the startup of the service will be fetched to memory. For on-demand images, the data has to be retrieved from the container registry. As shown in Fig. 9(b), for a 200 MB image, the service response latency for the traditional image format is at least 5 milliseconds at the 10 MB/s bandwidth, and the latency for on-demand is 17× to that of the traditional image. Compared to the traditional image that reads data locally, remote retrieving causes a long latency for on-demand images. For latency-sensitive services (e.g., web service), such a remote image data access mode inevitably compromises the performance of the function.

Container Running. This phase is from when the container starts to handle the request until the container is

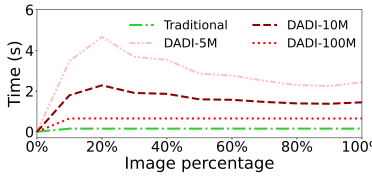
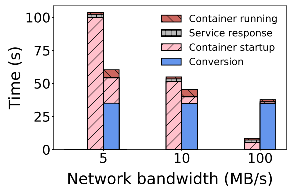


Fig. 10: The latency of accessing image data during container running. The x-axis represents the ratio of the amount of distinct accessed blocks relative to the image size. The y-axis shows access latency.

terminated. During the phase, the running service needs to access data in the container image. We vary the ratio of the distinct accessed blocks relative to the image size. As shown in Fig. 10, at a 5MB/s bandwidth, for a 140 MB image, the average access latency for the on-demand image is 3.1 seconds, approximately 20x higher than that for the traditional image. In addition, some data that the following requests will access is probably available locally. Thus, the latency for the on-demand image gradually decreases when a large amount of data has already been accessed. For services with large image sizes (e.g., ML), containing large amounts of data for training or inference, all the image data is likely to be downloaded locally incrementally as needed if the on-demand image is adopted.

It is important to note that during the deployment, we mainly focused on the network impact, because the impact of disk IO is slight. This is consistent with previous observations [3]. First, the disk I/O bandwidth is higher than the network bandwidth in real-world scenarios. Second, the disk write and download processes overlap with each other. We test the deployment times with HDD and SSD using *xwiki*, an image that involves a lot of IO access during deployment (about 200MB of files are accessed). The network bandwidth is set to 100MB/s. Deploying *xwiki* based on traditional images takes 45.2s and 43.8s for HDD and SSD, respectively. The time to deploy *xwiki* based on the on-demand image is 35.5s and 33.9s, respectively.

Image Conversion. For an on-demand image, users have to wait until the image is successfully converted from a traditional image before it can be pulled. Accordingly, we conduct an experiment to measure the overhead of updating an image (the testbed is an instance with Intel Xeon Platinum 8369B, 16 GB memory, and a 768 GB SSD with a


(a) Time overhead

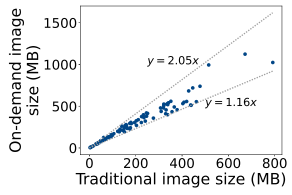
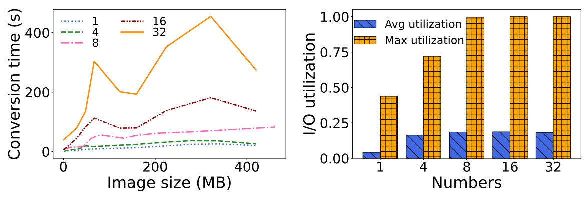

(b) Data size

Fig. 11: The simulation of image conversion



(a) Time overhead (b) Disk I/O

Fig. 12: The time overhead, and I/O utilization of conversion. Utilization is the ratio of the current bandwidth to the peak bandwidth. Average utilization is the ratio of the average bandwidth during the conversion to the peak bandwidth.

throughput of 750 MB/s). As shown in Fig. 11(a), the ratios of conversion overhead are 58%, 77%, and 93% for 5MB/s, 10MB/s, and 100MB/s, respectively. Furthermore, the converted image would be expanded in size compared to the traditional image. The converted on-demand image is 116%-205% to the corresponding traditional image (Fig. 11(b)).

Furthermore, a lot of system resources are consumed during image conversion. As shown in Fig. 12(a), the conversion time is influenced by the image size and the number of images to be simultaneously converted. When only one image needs to be converted, it takes about 7.0 seconds for a 100 MB image. The conversion time is 121.7 seconds when 32 images are converted together. As shown in Fig. 12(b), I/O utilization will reach 100% when 8 images are converted simultaneously. The machine's I/O bandwidth becomes a bottleneck for conversion. When it comes to DevOps mode that involves frequent uploads and pulls, using on-demand images can result in a high conversion overhead.

4.4 Key Observations and Implications

1. As computing paradigms and development modes evolve, images change significantly in size, and frequency of use:
   - Due to decoupling and customization, layers for emerging services (IoT, platform, and web services) tend to be smaller compared to traditional analytics services. For the existing container registry that manages layers uniformly, caching a layer of analytics may cause many small layers of other services to be improperly evicted. Therefore, layers of different types of services should be managed individually.
   - For services using DevOps mode, most images get cold rapidly, over 70% of such images staying active for less than 3 hours due to frequent updates. On the other hand, the images of services with traditional development mode tend to receive pull requests for a long period, with over 70% of them staying active for more than 11 days. When designing a cache for the container registry, we can make a quick eviction for web images that get cold rapidly to save cache.
   - For services that are decoupled or customized (IoT, platform, and web services), only a small portion of images are widely pulled. Therefore, P2P may not effectively accelerate most images for these emerging services. Even worse, the burden on the network may be aggravated by P2P requests.
2. We get some observations in layer-, image-, and file-level:
   - Although large layers are the minority, a large number of requests are for them. About 40% and 20% of requests are for layers larger than 10MB and 100MB, respectively. Large layers, which are overlooked by caching before, should be carefully cached as they take up large space, and some of them are frequently pulled.
   - Although the decoupling of services can provide elastic resource scaling, it also results in multiple discrete image pulls when deploying the service. The ccontainer registry can accelerate the deployment of a service by detecting and prefetching these associated images.
   - The file access pattern of a containerized service is related to the layer and image versions. For example, we find that the accessed files of containers are mainly distributed in upper layers, and services from the same repository exhibit similar access patterns. By leveraging this information, we can perform prefetching to optimize the deployment of on-demand images.
3. On-demand images are increasingly replacing traditional images, but they may not always be the optimal choice:
   - Using on-demand images requires data to be downloaded after services are started, resulting in more than 20 $\times$ response latency compared to traditional images. When image sizes are small, bandwidths are high, and the ratio of the accessed data to the entire image data is high, on-demand images may exhibit lower performance than traditional images.
   - On-demand images need to be converted from traditional images, and pull requests directly after uploads, which are quite common. Over 80% of the first pull is within 1 second after an upload, which cannot be served during the conversion. When deploying a container, it is important to select an appropriate image format according to service and network.

5 Optimization and Evaluation

We propose an image caching strategy according to the lifetime and association of images, named LACache, to speed up image provisioning. To select a suitable image format between the traditional and on-demand images for fast deployment, we design an image format recommend system, named IFRecommender.

5.1 Lifetime and Association-based Caching

In practice, the image layers are stored in the object storage service, and the combined cache (e.g., memory+SSDs) is widely used at container registries to hold frequently-accessed layers [5]. The layers pulled from the remote object store would be put in the cache to enable fast provisioning for subsequent requests.

To use the cache efficiently, we propose two enhancements based on previous observations. First, instead of caching small layers preferentially, large layers are also cached. To provide enough space, we categorize services and evict cached layers based on their type of service and lifetimes. Second, we leverage image-level prefetching based on the correlations among images to accelerate future pulls. Note that services are categorized based on their computing paradigm, development mode, and node performance, and are put into different namespaces.

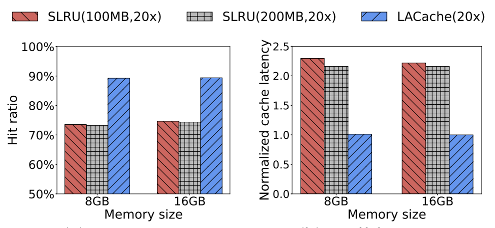
Evicting Cached Layers Based on the Lifetime. Through the previous observations, we find that the lifetimes of layers vary significantly in different services. For example, the lifetime of web services goes to only 10 minutes, far below the 300 minutes of IoT/ML. Therefore, the cached data can be evicted efficiently based on services. In particular, a service-based threshold is set to evict layers from memory. When the residency time of layers in memory exceeds the threshold of the services, the layers are moved into SSD. In this way, the out-of-date cached layers of each type of service can be cleaned out in time, and the memory will store more valid layers. Furthermore, if the long-lifetime data is incorrectly evicted, the high-capacity SSDs will continue to cache them.

Note that the threshold value for each type of service will be updated dynamically. Each type of service will be assigned a default value based on our analysis. When the cache space is insufficient, the image layer whose lifetime is greater than x and has not been accessed for y seconds will be evicted one by one according to the repositories' hotness from lowest to highest. When the available cache space continues to decrease, x and y are decreased until the cache space is sufficient. When there is a cache miss, it is checked if the image layer is incorrectly evicted due to the corresponding x and y, which will be adjusted respectively.

Prefetching Images Based on the Association. Considering that there are a large number of associated images, we can make use of the correlations to prefetch images from the object storage to the cache of the container registry to improve the hit ratio. In particular, we create an association list to record the image pairs. When an image is requested, its associated images will be put into the SSD based on the correlation characteristic. When the registry receives a pull request, the associations based on the user and time interval rules will be re-calculated. LACache ranks the image pairs based on their probabilities of pulling in pairs and prefetches the images with high probabilities of association. This design avoids triggering a chain of association pulls that could saturate the cache. Furthermore, The image pairs can be used to find complicated dependencies. For example, an e-commerce website has a long path. We can obtain this long path by finding three associated image pairs and combining them. It is noted that monolithic services, such as IoT and analytics, cannot be accelerated by this prefetch strategy, because there are very few dependencies in terms of function for these services.

Experiment Setup. The experiments are performed on a machine with 64GB memory and 768GB SSD. The object storage service is set on another node with the same configuration. The network bandwidth between nodes is 1000Mbps. The experiment workload is a dataset of pull traces for seven days, with 97394 requests tagged as web, or ML, or IoT. For LACache, the threshold for web, ML, and IoT are initialized to 1600, 1000, and 3000 seconds based on our observations.

Result Analysis. We compare LACache with the state-of-art work, SLRU [5]. SLRU puts the layers that are smaller than a certain threshold (e.g., 100MB and 200MB) in memory, and the large layers into SSD. And the replacement strategies in memory and SSD both are LRU. Fig. 13(a) shows the hit ratio of SLRU and LACache. Supposed that the size of memory is set as 8 GB, our algorithm improves the cache hit ratio by 22% on average. The hits of SLRU(100 MB, 20x) and LACache(20x) are 73% and 89%, respectively. And if the memory size is increased to 16GB, the hit ratio does not improve significantly. A small increase in the cache does not yield a substantial gain over large amounts of



(a) Hit ratio (b) Pull latency

Fig. 13: Comparison between SLRU and LACache in terms of cache hit ratio and the average latency for image pull

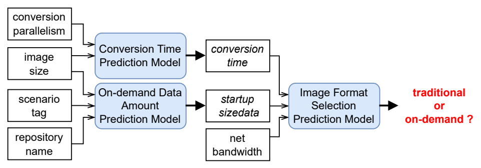
image data. SLRU(100 MB, 20x) and SLRU(200 MB, 20x) show little difference. As more layers are put into the memory, they are replaced more frequently. Thus, the SLRU which is a LRU strategy based on the size of the layer cannot effectively accelerate layer pulling.

Regarding the average latency of remote pull in all requests, LACache brings a great enhancement. As shown in Fig. 13(b), compared to SLRU(100 MB, 20x), SLRU(200 MB, 20x), the average pull latency is decreased by 57% and 54% respectively. The performance of 16 GB memory is similar to 8 GB. Compared to the caching strategy based on size, more requests in LACache will be pulled directly from the layered cache. At the current scale, SSD, HDD, and a small DRAM can achieve a high hit rate. When the scale of services further increases, NVM can be used to cache evicted layers from DRAM, like SSD. When utilizing NVM, the core concept of our design remains unchanged: evicting layers based on service type and prefetching images based on association.

5.2 Image Format Recommendation System

The deployment times of on-demand images are longer than that of traditional images when a large portion of image data are accessed and the pulls are immediately after the uploads under high bandwidth. For an image, it is necessary to predict its data access pattern and image conversion time based on its service type and category. The transmission times of on-demand images are determined based on the network and data access pattern, which are related to the service type. This information cannot be acquired by heuristic methods. IFRecommender collects relevant data in real-time and trains two neural network models online to predict the conversion time, and the amount of startup data for the on-demand image, respectively. Then, IFRecommender combines the two neural network's outputs and predicts a container image format with the shortest startup time. The architecture of IFRecommender is shown in Figure 14.

Model Architecture. To achieve the model rapidly, we use *Neural Network Intelligence* (NNI) toolkit [26] from Microsoft to automate its architecture. Each model in IFRecommender has three hidden layers, each with 64 neural nodes. For the output, the *argmax* function is used for two-class decisions. The prediction overhead is millisecond-level [26], which is negligible for image pulling.



conversion parallelism
image size
scenario tag
repository name
Conversion Time Prediction Model
On-demand Data Amount Prediction Model
conversion time
startup sizedata
net bandwidth
Image Format Selection Prediction Model
traditional or on-demand ?

Fig. 14: The architecture of IFRecommender

(**a**) **Conversion Time Prediction Model.** The conversion time required for the on-demand image is a major concern. From the previous analysis in Section 4.3, it is clear that the converted time of an image differs significantly when the size of the traditional image and the number of parallel converted images varies. Thus, we take these two factors as input features of the model. When a user requests an image, the model predicts the image conversion time based on the number of images currently being converted.

(**b**) **On-demand Data Amount Prediction Model.** For a container based on the on-demand image, the major overhead of startup is pulling the required data. From the previous analysis, the size of the startup data of one image is related to the size of the image. Furthermore, given that the startup data for images belonging to the same repository is similar, we also input the repository's name into the model to more accurately predict the size of startup data. It is customary to use the service name as the repository name for convenience. To prevent the misleading name of the repository, we can consider adding a feedback correction mechanism in the future. Also, we need to use category tag. Because in some scenarios, we do not have repository information, or the repository has few images. We can use category tag to provide some basic information. Thus, the model will predict the size of startup data for an on-demand image based on the repository name, category tag, and the size of the on-demand image.

(**c**) **Image Format Selection Prediction Model.** IFRecommender first obtains user's available network bandwidth. Then, the results predicted by model **a** and model **b** will be used as inputs to predict the time overhead of two different image formats, as shown in Figure 14. Accordingly, the image format with the minimum time overhead is recommended to the user. Note that, for an available on-demand image, its conversion time is 0 and the startup data size can be obtained directly.

Collecting and Labeling Data. To collect data, we practically evaluate the deployment times of 500 images in two formats. Specifically, the images are tested under 50 different bandwidths (ranging from 1Mbps to 100Mbps), and 5 different conversion parallelisms (i.e., 1, 2, 4, 8, 16). Each example in the dataset is represented by multiple fields, <*image_size, category_tag, repository_name, conversion_parallelism, conversion_time, converted_size, net_bandwidth, startup_data_size, optimal_image_format*>. If the performance of the traditional image outperforms that of the on-demand, *optimal_image_format* is set to 1. Otherwise, it is 0. Note that 80%, 10%, and 10% of the dataset are used for training, cross-validation, and testing, respectively. In the testing dataset, IFRecommender's accuracy reaches 97%.

Performance Evaluation. We replay all the traces to test the image deployment latency for four deployment methods: 1) all images deployed in the traditional format; 2) all images deployed in the on-demand format; 3) images deployed in the format predicted by IFRecommender; 4) images deployed in ideal format. We respond to each image pull by separately pulling an on-demand image and a traditional image, and then selecting the format with a shorter deployment time for each request as the ideal format. Fig. 15 shows the results. We can see that the latency of deploying with on-demand images is less than that of traditional images, regardless of categories. The on-demand image format has an evident advantage due to the significant reduction of data pulling. Besides, deployment latency via IFRecommender is lower than that of the other two in all categories. Especially for web and platform services, compared to the on-demand image, the deployment latency is reduced by 39% and 42%, respectively. The reason is that there are many images involving frequent DevOps operations and lots of I/O of image data. In these cases, the performance of the traditional image is better than that of on-demand images. Our method shows 6.5%, 1.2%, 4.4%, 5.6%, and 3.8% longer deployment time for analytics, IoT, ML, web, and platform services, respectively, compared to the ideal situation. This means that IFRecommender can make the right decisions in most cases.

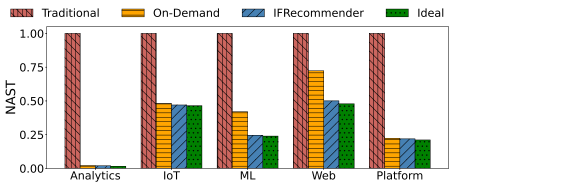


Fig. 15: Comparison of three deployment methods. *Normalized average startup time* (NAST) refers to the ratio of the average startup time to that of all services deployed by traditional images.

Note that the design of IFRecommender is straightforward in terms of the inputs and the model architecture. Furthermore, IFRecommender is deployed in the container registry, and the memory and performance information of the client side is unavailable. Naturally, researchers can further optimize the models by further analyzing the image traces or moving it to the client side.

6 Related Works

Analysis of Container Image Request. In terms of layer request analysis, Anwar et al. [5] find that most pull requests mainly concentrate on a small portion of layers. Albahar et al. [6] analyze layer requests from the perspective of users and find that a user usually downloads an image only once or repeatedly. Ahmed et al. [27] analyze the influence of the parallelism of image transfer and decompression when downloading images. Zhao et al. [13] find that network bandwidth is the bottleneck for smaller layers. In terms of file access analysis, Harter et al. [3] observe file access amount in the process of deploying containers. Gkikopoulos et al. [28] analyze the dependency between image files when special hardware is used. Xu et al. [29] observe the performance of different storage and storage drivers. Wu et al. [30] analyze the latency and contention in storage drivers. Existing works mainly analyze the content of traditional images and lack analysis of different types of services.

Analysis of Image Content. For layers of images, Sharma et al. [31] find that container images have smaller sizes compared to VM images. Funari et al. [4] observe the distribution of images in clusters and find that as the number of nodes increases, the effectiveness of layer-level sharing decreases. Zhao et al. [12] analyze the layer redundant in the registry and find that images share numerous small-sized layers. For files inside layers, Gholami et al. [14] find that updating software packages in image layers directly may cause stability issues. File access patterns during containers' run are essential.

Optimizations on Image Transfer. There are three main ways to optimize data transfer: 1) On-demand image. Slacker [3] is the first on-demand image based on NFS. DADI [2] organizes and transmits image data in the granularity of block. FogDocker [18] analyzes required files when running a container based on Dockerfile and builds a small image for deployment. Gear [9] and Nydus [10] realize index-based on-demand file download; 2) P2P transfer. Dragonfly [7] is a P2P-based layer transfer method that dynamically divides and compresses layers based on bandwidth. FaaSNet [8] design a self-balancing binary tree topology to achieve efficient P2P image download while reducing network congestion; 3) Data sharing. To reduce the amount of data to be downloaded, Wharf [32] and Cider [33] realize layer-level sharing within clusters. ADAL [34] tries to find nodes with the most shared layers to deploy containers. Cntr [35] redivides the image so that containers can be launched as soon as possible. FastBuild [36] builds a local cache to reduce repeated data downloads when building images.

Container Registry. Bolt [37] realizes efficient image layer distribution and searches in the registry cluster based on hash strategies. Anwar et al. [5] design a hotness-based layer cache. Albahar et al. [6] design a prefetch strategy based on the dependency of layers for fast layer provision.

7 Conclusion

Efficient image management is critical to the rapid response and efficient maintenance of container cloud platforms. To better understand the characteristics of container images, we design a toolkit, named *mTracer*, which can capture operational traces at different granularities. With *mTracer*, we collect real-world traces and analyze the characteristics of images in different types of services (i.e., analytics, IoT, ML, web, and platform services). We obtain some different and fresh observations. Based on our observations, we propose two optimizations to clarify the practicality of our observations. The lifetime and association-based caching strategy can reduce layer pull latency by up to 57%, and the neural network-based image format recommendation system can reduce image deployment latency by up to 42%.

8 Acknowledgement

This work was supported in part by the National Key Research and Development Program of China under grant 2022YFB4500704 and in part by the National Science Foundation of China under grants 62032008, 62232012, and 62202196.

References

[1] "Docker hub," https://hub.docker.com/, 2022.

[2] H. Li, Y. Yuan, R. Du, K. Ma, L. Liu, and W. Hsu, "Dadi: Blocklevel image service for agile and elastic application deployment," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020, pp. 727–740.

[3] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 181–195.

[4] L. Funari, L. Petrucci, and A. Detti, "Storage-saving scheduling policies for clusters running containers," *IEEE Transactions on Cloud Computing (TCC)*, pp. 1–12, 2021.

[5] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebr, and A. R. Butt, "Improving docker registry design based on production workload analysis," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2018, pp. 265–278.

[6] 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 *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020, pp. 769–783.

[7] "Alibaba Dragonfly," https://d7y. io/en-us/, 2022.

[8] 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 *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2021, pp. 443–457.

[9] 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 *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 115–125.

[10] "Nydus," https://github.com/dragonflyoss/image-service, 2022.

[11] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *Proceedings of the USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, 2020, pp. 1–7.

[12] 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 *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)*, 2019, pp. 1–10.

[13] 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 (TPDS)*, vol. 32, no. 4, pp. 918–930, 2021.

[14] S. Gholami, H. Khazaei, and C. Bezemer, "Should you upgrade official docker hub images in production environments?" in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2021, pp. 101–105.

[15] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems (Eurosys)*, 2015, pp. 115–125.

[16] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo, "Carving perfect layers out of docker images," in *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019, pp. 1–6.

[17] "Open Container Initiative," https://opencontainers.org/, 2022.

[18] L. Civolani, G. Pierre, and P. Bellavista, "Fogdocker: Start container now, fetch image later," in *Proceedings of IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2019, pp. 51–59.

[19] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[20] "Containers on IBM Cloud," https://www.ibm.com/cloud/containers, 2023.

[21] "Azure," azure.microsoft.com/en-gb/, 2023.

[22] "Containers on AWS," https://aws.amazon.com/containers/services/n=ch_ls, 2023.

[23] X. You, C. Zhang, X. Tan, S. Jin, and H. Wu, "AI for 5g: research directions and paradigms," *Science China Information Sciences*, vol. 62, no. 2, pp. 1–13, 2019.

[24] "Linux Runtime Security and Forensics using eBPF," https://github.com/aquasecurity/tracee, 2022.

[25] "NS3," https://www.nsnam.org, 2023.

[26] "Microsoft Neural Network Intelligence," https://github.com/Microsoft/nni/, 2022.

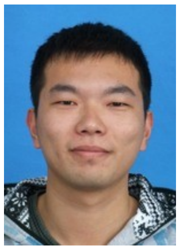
[27] A. Ahmed and G. Pierre, "Docker container deployment in fog computing infrastructures," in *Proceedings of the 2018 IEEE International Conference on Edge Computing (ICEC)*, 2018, pp. 1–18.

[28] P. Gkikopoulos1, V. Schiavoni, and J. Spillner, "Analysis and improvement of heterogeneous hardware support in docker images," in *Proceedings of the Springer International Federation for Information Processing (IFIP)*, 2021, pp. 125–142.

[29] Q. Xu, M. Awasthi, K. Malladi, J. Bhimani, and M. Annavaram, "Performance analysis of containerized applications on local and remote storage," in *Proceedings of the International Conference on Massive Storage Systems and Technology (MSST)*, 2017, pp. 1–12.
[30] S. Wu, Z. Huang, P. Chen, H. Fan, S. Ibrahim, and H. Jin, "Container-aware i/o stack: bridging the gap between container storage drivers and solid state devices," in *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2022, pp. 18–30.
[31] P. Sharma, L. Chaufournier, P. J. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: a comparative study," in *Proceedings of International Middleware Conference (Middleware)*, 2016, pp. 1–13.
[32] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain, M. Mohamed, D. Skourtis, A. Warke, and D. Hildebrand, "Wharf: Sharing docker images in a distributed file system," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2018, pp. 174–185.
[33] L. Du, T. Wo, R. Yang, and C. Hu, "Cider: a rapid docker container deployment system through sharing network storage," in *Proceedings of the IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2017, pp. 332–339.
[34] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2021, pp. 1–9.
[35] J.Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight os containers," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018, pp. 199–212.
[36] Z. Huang, S. Wu, S. Jiang, and H. Jin, "Fastbuild: Accelerating docker image building for efficient development and deployment of container," in *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2019, pp. 28–37.
[37] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. Butt, "Bolt: Towards a scalable docker registry via hyperconvergence," in *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, 2019, pp. 358–366.

Zhuo Huang received the PhD degree from Huazhong University of Science and Technology (HUST) in 2023. Currently he is working as a post-doctor in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL), Huazhong University of Science and Technology (HUST) in China. His current research interests include container virtualization, serverless computing optimization, and storage system.

Qi Zhang received his B.S. degree from the Huazhong University of Science and Technology in 2021 and is currently pursuing his M.S. degree in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL), HUST. His research interests include operating systems, performance evaluation, and lightweight virtualization technologies.



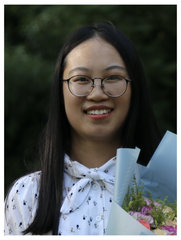
Hao Fan received the PhD degree from Huazhong University of Science and Technology (HUST) in 2021. Currently he is working as a post-doctor in Service Computing Technology and System Lab (SCTS) and Cluster and Grid Lab (CGCL), Huazhong University of Science and Technology (HUST) in China. His current research interests include container technology and storage system.

Song Wu received the PhD degree from Huazhong University of Science and Technology (HUST) in 2003. He is a professor of computer science at HUST in China. He currently serves as the vice dean of the School of Computer Science and Technology and the vice head of Service Computing Technology and System Lab (SCTS) and the Cluster and Grid Computing Lab (CGCL) in HUST. His current research interests include cloud resource scheduling and system virtualization. He is a member of IEEE.

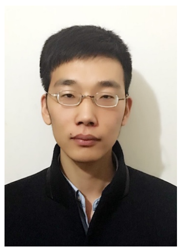
Chen Yu received the PhD degree in information science from the Tohoku University in 2005. From 2005 to 2006, he was a Japan Science and Technology Agency postdoctoral researcher with the Japan Advanced Institute of Science and Technology. He is with the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), where he is currently a professor working in the areas of cloud computing, ubiquitous computing, and green communications. He is a member of IEEE.

Hai Jin Hai Jin is a Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. Jin received his PhD in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is a Fellow of IEEE, Fellow of CCF, and a life member of the ACM. He has co-authored more than 20 books and published over 900 research papers. His research interests include computer architecture, parallel and distributed computing, big data processing, data storage, and system security.

Jun Deng is with Alibaba Cloud. He received the B.E. degree from Zhejiang University of Technology in 2010. His research interests include cloud computing, distributed system and cloud native architecture.



Jing Gu is with Alibaba Cloud. She received a master's degress from Peking University in 2019. Her research focuses on cloud computing, distributed system and AIOps.



Zhimin Tang received the BS degree in Information Management and Information Systems from Nanjing University in 2008. He is a senior staff engineer in Alibaba cloud, Hangzhou, China. His research interests include cloud computing, serverless and distributed system.