Quicklayer: A Layer-Stack-Oriented Accelerating Middleware for Fast Deployment in Edge Clouds

Yicheng Feng
Tianjin University
Tianjin, China

Shihao Shen
Tianjin University
Tianjin, China

Cheng Zhang
Tianjin University of Finance and Economics
Tianjin, China

Xiaofei Wang*
Tianjin University
Tianjin, China

ABSTRACT

Containers are gaining popularity in edge computing due to their standardization and low overhead. This trend has brought new technologies such as container engines and container orchestration platforms (COPs). However, fast and effective container deployment remains a challenge, especially at the edge. Prior work, which was designed for cloud datacenters, is no longer suitable for container deployment in edge clouds due to bandwidth limitations, fluctuating network performance, resource constraints, and geo-distributed organization. These edge features make rapid deployment on the edge difficult. Additionally, integrating with COPs is crucial for successful deployment.

We present Quicklayer, a layer-stack-oriented middleware designed to accelerate container deployment in edge clouds. Quicklayer takes a holistic approach that preserves the stack-of-layers structure and is backward-compatible. It includes (1) a layer-based container refactoring solution that optimizes container images while maintaining the layer structure, (2) a customised Kubernetes scheduler that is able to be aware of network performance, disk space, and container layer cache for container placement, and (3) distributed shared layer-stack caches which are optimized for cooperative container deployment among edge clouds. Preliminary results indicate that Quicklayer reduces redundant image size by up to 3.11× and speeds up the deployment process by up to 1.64× compared to the current popular container deployment system.

CCS CONCEPTS

• **Networks** → *Network management*; *Cloud computing*.

KEYWORDS

Container, Fast deployment, Distributed cache, Registry, Kubernetes, Scheduling, Edge computing

*Corresponding Author: Xiaofei Wang (xiaofeiwang@tju.edu.cn).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

APNET 2023, June 29–30, 2023, Hong Kong, China

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0782-7/23/06…$15.00
https://doi.org/10.1145/3600061.3600074

ACM Reference Format:
Yicheng Feng, Shihao Shen, Cheng Zhang, and Xiaofei Wang. 2023. Quicklayer: A Layer-Stack-Oriented Accelerating Middleware for Fast Deployment in Edge Clouds. In *7th Asia-Pacific Workshop on Networking (APNET 2023), June 29–30, 2023, Hong Kong, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3600061.3600074

1 INTRODUCTION

Container engines, such as Docker [2] and containerd [8], have proven advantages in standardization, user-friendliness, and low overhead [17, 18]. Compared to virtual machines (VMs), containers provide lightweight isolation with new DevOps features like incremental updates [31]. Application packaging is simplified by creating container images that can be uploaded to a registry, such as Docker hub [3], for storage or sharing. Deployment is also straightforward: pull images from the registry and start the container process.

Container orchestration platforms (COPs) such as Kubernetes (K8s) [4], Docker Swarm [5], Apache Mesos (with Marathon) [1], are developed to manage containers by automating application deployment and coordinating resource allocation between containers. Container engines and COPs are often used together, for example, with operators preferring Docker for building and pushing containers to the registry and K8s for large-scale deployment and upgrades of containers, such as Amazon EKS[11], Microsoft AKS [13], and Google GKE [10].

While container technologies were initially designed for cloud datacenters [6], they are now gaining popularity in edge computing. Edge clouds use industry-standard hardware to offer containerized services to end-users organized by COPs [25]. These services benefit from reduced latency, power consumption, and bandwidth usage, as they are located closer to users [29].

Fast container deployment [1] is crucial in modern edge computing environments [17]. Slow deployment can lead to SLA violations, such as poor responsiveness in serverless computing [26]. Fast container deployment in edge clouds faces unique challenges. First, high latency and limited bandwidth can slow down the process of pulling container images from remote registries [17, 19]. Second, the variable network performance and geo-distributed nature of edge clouds make it difficult to place containers effectively, further slowing down deployment. Finally, limited resources in edge clouds

[1]We define the container deployment as the process which includes container image downloading, layers extracting, and container runtime starting.

make the current cache solutions, which use complete container images as storage granularity, too expensive.

Previous work has proposed solutions to accelerate container deployment [19, 31, 35]. However, these solutions are not well-suited for fast deployment in today's edge clouds. Solutions like FaaSNet [31] and Wharf [36] assume good network conditions, while in reality, upstream links between the edge cloud and the remote site often have poor bandwidth and high latency [17]. Although Starlight [17] provides edge-specific solutions, there is still a lack of consideration for a complete pipeline for container deployment, including container placement, which can have a significant impact on rapid deployment in edge clouds.

The stack-of-layers structure is a fundamental design for containers [17, 27, 32]. In contrast to previous work [18, 26], we contend that a well-considered application of this structure, such as a comprehensive solution, can still support fast container deployment in edge clouds. Our analysis of 10,000 edge clouds in the wild [2] demonstrates that there is still significant potential for better utilization of disk resources in edge clouds and upstream link bandwidth between nearby edge clouds (§2.3). Driven by these insights, we propose Quicklayer, a middleware that leverages the layer-stack structure to accelerate container deployment in edge clouds. Quicklayer offers a holistic solution around the stack-of-layers structure without modifying operational and development pipelines. Quicklayer can be activated through the original K8s CLI command via non-intrusive integration with K8s. We summarize our contributions as follows:

- We propose Quicklayer, a layer-stack-oriented acceleration middleware for fast container deployment in edge clouds. Quicklayer fully exploits the potential of edge clouds and provides a holistic approach around the layer-stack structure to accelerate deployment.
- We design a image refactoring solution which is compatible with all standard container engines and registries. It optimizes images and preserves the convenient stack-of-layers structure of containers.
- We implement a customized K8s scheduler which extends the awareness of network performance, disk space, and container layer cache to make a suitable container placement for fast deployment.
- We design a distributed shared layer-stack cache and make cooperative container deployment among edge clouds to accelerate deployment.

Our preliminary results in testbed show that Quicklayer can reduce the redundant size of images by up to 3.11× and speed up the deployment process by up to 1.64× compared to the current popular container deployment system.

2 BACKGROUND AND MOTIVATION

2.1 Container and K8s

Container. A *container image* is a template for creating a container, which consists of two main parts: container configuration metadata and a sequence of layers. Concretely, the container configuration metadata is an overview of the entire container, recording the identity information as well as the layer digest (an SHA256 hash of the layer's contents) it contains and the parent-child relationship between layers. Each layer is comprised of files and their associated metadata. In general, when a container image is pulled from a registry, it will be launched into the root filesystem by a graph driver. To start a container instance, a writable layer will be created and a standard runtime is used to start the container process.

[2] PPIO Edge Cloud (www.ppio.cn) supports for the production dataset.

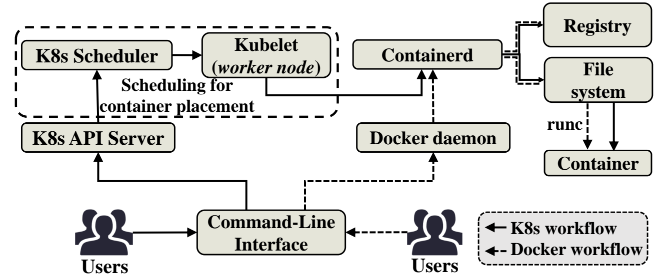

Figure 1: The deployment flow of K8s and Docker.

K8s. K8s is now the representative of COPs. Most edge and cloud platforms use K8s to deploy and maintain their containerized services instead of directly using a container engine [10, 11, 13]. Currently, K8s uses *containerd* [8] as the internal container engine while Docker support has been phased out since K8s v1.20. It indicates that solutions [18, 26] adhered to Docker (e.g., dependent on Docker Daemon) probably cannot work. Figure 1 shows a different deployment workflow from using Docker directly. Therefore, solutions that focus only on container engines do not take into account the full pipeline of COPs, especially container placement, which has a non-negligible impact on deployment time. For example, edge clouds with good network performance and cache probably enable faster container deployment.

2.2 Why Container Image Refactoring?

Container layer-based structure rethinking. The stack-of-layers design is undoubtedly one of the key features of containers [17, 27, 32]. It provides a standardized I/O stack granularity that streamlines container development and minimizes redundancy across layers and containers by referencing the layer with the same digest. However, this design also presents a challenge for container deployment acceleration, as there is significant redundancy across different digest layers [34]. Docker Hub analysis reveals that over 99.4% of files contain duplicates [33], slowing down container image transfers and placing a strain on bandwidth and storage capacity [17, 18].

Paradoxically, the introduction of a new format or modification of the container image granularity, such as adopting a file-based structure for on-demand downloading [22, 26], must carefully consider its impact on the entire deployment and operation pipelines, as it may introduce new devices and file systems, making backward compatibility challenging. Furthermore, a more granular deduplication solution can potentially impose additional overhead in terms of latency and maintenance [26, 32]. For instance, a simplistic file-based structure solution can result in up to 98× higher layer pull latency compared to a deduplication-free registry due to the larger number of individual file objects [32].

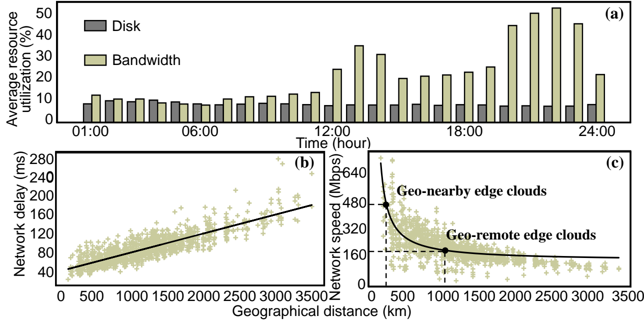

Figure 2: Measurements on (a) resource utilization, (b) round-trip times (RTT), and (c) upstream bandwidths.[2]

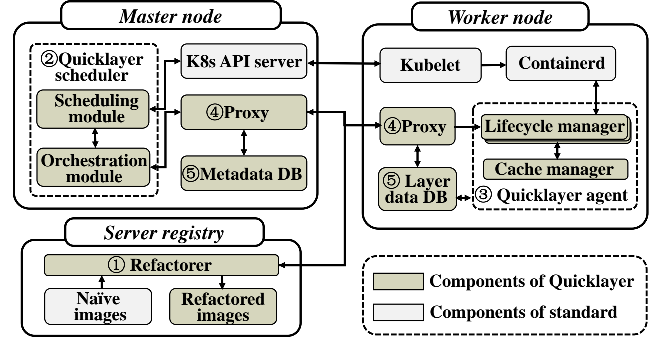

Figure 3: The architecture of Quicklayer.

Therefore, Quicklayer aims to find a compromise by proactively refactoring the entire container image, while preserving the stack-of-layers structure (§3.2). [3] This approach is backward compatible with the Open Container Initiative (OCI) image specification [7], which ensures that no new devices or file systems are introduced, while improving the efficiency of container deployment by reducing redundancy.

2.3 Why Layer-stack Cache Sharing?

It is reported that container image download accounts for about 80% of the total container deployment time [30]. Therefore, to speed up deployment, reducing the download latency is especially important. One typical approach is to provide better networking, getting container images from closer to the edge cloud or even locally cached.

Great potential in edge clouds. Based on our measurement over 10,000 edge clouds in the wild,[2] we found that the average utilization of the disk and bandwidth resource are 8.13% and 21.69% (see Figure 2), which remains significant potential for improvement. Meanwhile, the geo-nearby edge clouds show much better network performance than the remote. For example, an edge cloud 280km apart has three times the upstream bandwidth of one 1000km apart, which is about 480Mbps.

Distributed layer-stack cache sharing. Based on our analysis of the workload dataset from IBM [16], we have observed that more than 50% of the total requests are directed towards the top 1% of the most popular containers, highlighting the significance of caching in container deployment scenarios from a practical application standpoint. It is worth noting that, by default, most nodes maintain a cache of container images after pulling them from the registry. However, in resource-constrained edge cloud environments, employing a complete image as the caching granularity proves to be prohibitively expensive. Therefore, in the context of high reuse of frequently accessed layers (e.g., base layers like CentOS), particularly following container refactoring, it is more appropriate to cache these hotness layers.

Taking inspiration from this observation, Quicklayer leverages a shared layer-stack cache (§3.3) and facilitates cooperative deployment by facilitating layer transfers among geographically proximate edge clouds. One common concern pertains to the feasibility of shared layer transfers among edge clouds. Although the presence of NAT networks makes establishing direct peer-to-peer (P2P) connections between edge clouds challenging, this issue can be effectively addressed through the application of TCP/UDP hole-punching techniques [23, 24]. This approach enables efficient layer transfers between two edge clouds, ensuring the viability of the shared layer transfer mechanism.

Customized scheduler. Although K8s can organize geo-nearby edge clouds into a cluster, it cannot sense layer-stack cache and network performance in edge clouds when scheduling, leading to inappropriate container placement. Moreover, optimal management of layer-stack caches and collaborative transfers between geo-nearby edge clouds are also missing. Quicklayer addresses these issues by implementing a *customized scheduler* (§3.4).

3 DESIGN

3.1 Design Overview

The architecture of Quicklayer is shown in Figure 3, with three main roles: the *server registry*, the *master node*, and the *worker node*. A *K8s edge cloud cluster* comprises at least one *master node* and several *worker nodes*. The master node is responsible for controlling the worker nodes and deploying containers for request processing. We describe the main components of Quicklayer below.

The *refactorer* ① is responsible for periodically evaluating the redundancy of the registry and determining whether to refactor container images. When a refactoring event occurs, the metadata and layers in the standard registry are modified (with a template copy created to prevent service disruption). The refactorer also synchronizes the latest container metadata to the edge cloud and cleans invalid cache.

The *Quicklayer scheduler* ② consists of a *scheduling module* and an *orchestration module*. The scheduling module makes the placement decision for deployment events by considering various metrics and notifies the orchestration module of the scheduling result. The orchestration module provides a cooperative deployment solution, and the *proxy* ④ sends the deployment manifest (tasks

[3]Quicklayer's deduplication can achieve the same effect as file-based deduplication by proactively restructuring the container based on redundant file content.

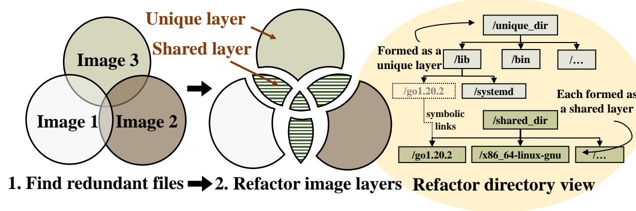

Figure 4: Overview of container image refactoring.

for each node generated by the orchestration module) to the edge clouds.

The *Quicklayer agent* ③ focuses on two main tasks. The *lifecycle manager*, a built-in module, asynchronously restores incoming distributed image layers to regroup them into a complete container image at the binding node and initiates the loading step. Once the filesystem loading process is complete, blocked K8s deployment instructions will resume, and the container will start. The *cache manager*, another module, dynamically optimizes the layer-stack cache using a cache replacement algorithm.

A *proxy* ④ is not a simple bridge but is responsible for synchronizing cache distribution from *worker nodes* to *master nodes*. It opens an HTTP connection, fetches and sends layer data or metadata based on the deployment manifest from the database, and downloads and decompresses the file to the appropriate directory.

Two *DBs* (metadata DB and layer data DB) ⑤ cache the metadata and layer data of the container images. They are stored separately based on the nodes' identity, and copies of caches are transferred to the designated edge cloud when needed or create hard links to avoid multiple writes to the underlying filesystem.

3.2 Container Image Refactoring

Quicklayer designs an effective refactoring solution that removes the redundancy at the file level while maintaining the structure at the layer level. Note that the refactoring solution follows the OCI image specification [7], therefore, it is applicable to any image format conforming to the OCI specification.

Refactoring preparation. The refactorer component in the registry server performs periodic detection of image redundancy based on container configuration metadata, encompassing three key steps: (1) retrieving path, name, size, and SHA256 hash value of each file; (2) iterating through the metadata and constructing a mapping table that indexes image IDs, enabling multiple image IDs to be associated with the metadata of common files; and (3) calculating the current redundant file size for each image using the index table. To enhance user convenience, a user-friendly event trigger is provided in the form of a constant threshold value for redundancy size. When the image redundancy size within the registry server surpasses the threshold, the refactorer initiates image refactoring operations.

Image refactoring. Figure 4 depicts a comprehensive diagram of the refactoring process. Initially, redundant files identified through the preparatory refactoring stage are relocated to a shared directory, leveraging the mapping table. Subsequently, the files within each refactored image are divided into two segments: a unique portion and a shared portion. To preserve the original file calling relationships, symbolic links are generated for the files moved to the shared directory. Finally, the hash value of the contents is used to calculate the digest of each unique and shared directory. Quicklayer reconstructs and organizes layer data and metadata to form a complete image. To prevent excessive layer division, only redundant files exceeding a user-defined threshold (on-demand) are moved to the shared directory. The resulting refactored image can seamlessly function with container engines, standard repositories, and remains transparent to users.

Complexity analyse. The refactoring algorithm has a complexity of $O(n)$. It involves iterating through n files to build the mapping table and iterating through the mapping table to refactor the layer structure.

3.3 Distributed Shared Layer-stack Cache

Quicklayer uses a layer as the cache granularity instead of the image to optimize cache efficiency. This is because edge clouds have limited resources, and the stack-of-layers structure is the standard structure of container images in the container I/O stack. Moreover, organizing the cache in layers reduces redundancy, especially after container image refactoring. Quicklayer organizes the layer-stack cache in the edge cloud using a directory tree structure. Users can set the caching space size while initiating the custom scheduler by applying K8s scheduler's YAML configuration.

Cache optimization. To optimize cache hit ratios, Quicklayer uses the Adaptive Replacement Cache (ARC) replacement policy [28], which keeps track of both the recently and frequently used layers and adapts to changing access patterns. In the early stages, the cache space in each edge cloud gradually fills up with image layers pulled from the registry. Once the cache space is full, frequently and recently used layers are prioritized as caches, and some of the used layers are replaced to free up space. This approach differs from the K8s default cache policy (i.e., garbage-collection [9]). When a local deployment event occurs, the caches are symbolic linked to a temporary folder for container restoring and loading. To ensure adequate fault tolerance, the replacement of the involved caches is deferred until the container images have finished loading.

3.4 Customized K8s Scheduler

As shown in Figure 5, we follow the open source best practices [21] to implement the customized K8s scheduler.

Scheduling module. Through the native CLI command of K8s, users can initiate container deployment events, which are then processed by various components within the K8s ecosystem. The *K8s API Server* communicates the deployment event to the *customized scheduler*, which performs scheduling based on a multi-step process consisting of sorting, filtering, scoring, and binding. This process is executed by a set of plugins, and the node with the highest score is selected for container deployment. [4] In the context of Quicklayer, the scheduling module maintains the multi-staged design while enhancing and extending several plugins, as outlined in Table 1.

[4] In K8s, the scheduling process revolves around the pod object, which can comprise multiple containers. For the purpose of clarity and simplicity, we will refer to these pods as "containers" throughout this paper.

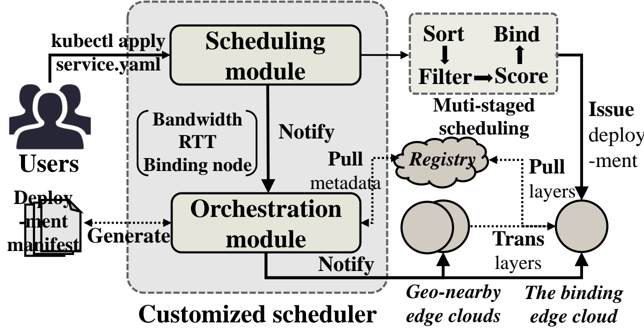

Figure 5: The workflow of customized K8s scheduler.

Table 1: Extra and modified plugins in scheduler.

Plugin	Kind	Weight	Stage
Network performance	Extend	1	Filtering & Scoring
Layer locality	Extend	1	Scoring
Resources balanced allocation	Modify	1	Scoring
Least requested priority	Modify	1	Filtering & Scoring

The *Network Performance plugin* calculates the node's network performance score by considering upstream bandwidth and round trip time, which are measured by a dedicated measurement pod. Given the resource constraints of edge nodes, disk over-occupation can significantly impact node stability. Therefore, we have integrated the *ephemeral storage* metric into the *Resources Balanced Allocation* and *Least Requested Priority plugins*. Furthermore, we have replaced the *Image Locality plugin* with a more granular plugin called *Layer Locality*, which extends the awareness of layer caching. To mitigate node heating issues [12], we compute the Layer locality score as follows:

$$LayerLocality = cacheHitScore \cdot spread + cacheFreeScore.$$

The *cacheFreeScore* and *spread* factor are employed to prevent concentrated deployment events on the same node. Once the scheduling process is completed, the module provides information regarding the binding node and the network performance of cluster nodes to the orchestration module. Simultaneously, the deployment event is forwarded to the *Kubelet* of the binding node.

Orchestration module. Upon the occurrence of a deployment event, the orchestration module initiates a validation process to determine if the container is cached within the edge cloud cluster, based on the user-submitted application YAML configuration. To facilitate a cooperative transfer solution for deployment, the module generates a deployment manifest that incorporates (1) metadata obtained from the metadata DB or retrieved from a remote registry, and (2) information provided by the scheduling module. This manifest includes transfer tasks assigned to each edge node, with a preference given to nodes exhibiting superior network performance for layer transfers. For instance, if the binding node "node1" requires "layer1" to reconstruct the container image, the module instructs "node2," an edge cloud within the cluster that caches "layer1" and possesses a favorable network performance, to transmit the layer to "node1." In cases where the required "layer1" is unavailable

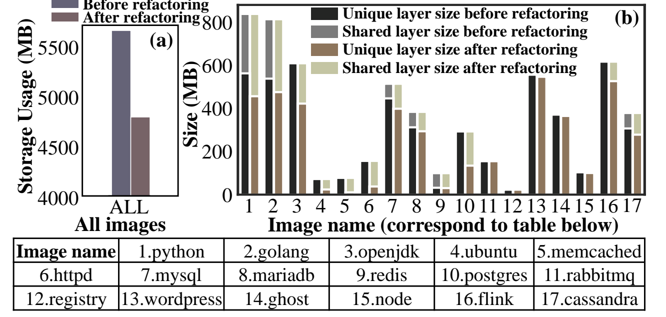

Image name	1.python	2.golang	3.openjdk	4.ubuntu	5.memcached
6.httpd	7.mysql	8.mariadb	9.redis	10.postgres	11.rabbitmq
12.registry	13.wordpress	14.ghost	15.node	16.flink	17.cassandra

Figure 6: Overall storage saving by refactoring (a) and the comparison of image composition (b).

within the cluster, or if all other node networks are experiencing suboptimal performance, "node1" is directed to request the layer from the registry server via the registry REST API. To minimize the latency introduced by round-trip requests, the deployment manifest is sent by the master node rather than the binding node. For practical considerations, the concurrent upload connection setting is limited to three connections.

4 PRELIMINARY EVALUATION

4.1 Experimental Setup

Testbed setup. We conduct our evaluation on a testbed comprising two edge cloud clusters, each consisting of one master node and four worker nodes. Each worker node is equipped with 2 cores (vCPUs, 2.20GHz Intel Xeon E5-2630) and 4GB RAM, while the master node and the registry server are configured with 4 vCPUs and 8GB RAM. We deploy the latest release of Kubernetes v1.24.10 on the edge cloud cluster, and Docker Registry 2.0 v2.8.1 is used as the standard registry on the registry server. To control the bandwidth, we utilize the Linux Traffic Control (TC) tool [15]. We limit the bandwidth to 400Mbps within the edge cloud cluster and 100Mbps between the cluster and the remote registry server, based on our measurement presented in §2.3.

Containers and workloads. We evaluate Quicklayer using a set of 17 popular official images totalling 5.96GB from the Docker Hub [3]. For our experiments, we use a real workload dataset from IBM [16], where the "timestamp" in the dataset is considered as the request arrival time, and the "http.request.uri" is considered as the container type to determine the frequency and type distribution of container deployment requests.

4.2 Preliminary Results

In Figure 6(a), we compare the total storage size of the 17 container images in the registry before and after refactoring with Quicklayer. The results show that Quicklayer effectively reduces the redundant size of images by up to 3.11×, saving 15.5% of storage space. This is because Quicklayer significantly increases the proportion of shared layers in a container image (see Figure 6(b)), indicating that more layers will be reused by other images. It is worth noting that Quicklayer's container image refactoring not only reduces image storage space in the registry but also benefits from a series of deployment

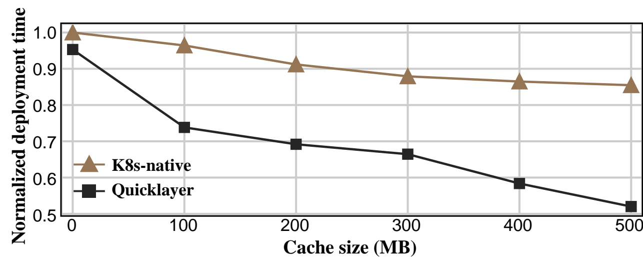

Figure 7: Container deployment time comparison under different cache size.

processes, such as download and load, while retaining the container key design of the stack-of-layers structure.

To evaluate the acceleration of Quicklayer on container deployments, we set the same cache size for the baseline, which uses the current popular container deployment system, K8s. As shown in Figure 7, Quicklayer performs well at multiple cache size settings, even when it is set to 0MB. This is due to the more lightweight deployment process supported by Quicklayer's container image refactoring, a more suitable container placement selected by Quicklayer's customized K8s scheduler, and a shared layer cache that makes deployment cooperative among the 4 geo-nearby edge clouds organized in a K8s cluster. Quicklayer speeds up the container deployment process by up to 1.64× compared to the baseline with 500MB cache space, and the layer hit ratio is 39.7% higher than the default lagging image cache.

5 RELATED WORK

Container registry. DupHunter [32] designs a Docker registry architecture to deduplicates layers and restore overhead. Li et al. [27] propose a reconstruction algorithm for Dcoker images in simulation. Quicklayer also remove redundancy by container image refacotring, but it's proven to be compatible with standard container engines and registries.

On-demand downloading. Slacker [22] utilizes NFS to enable on-demand downloading of required data. Similarly, DADI [26] uses on-demand fetching by operating at the block level, which also needs a tailored image format and registry. Quicklayer preserves the convenient layer structure through refactoring and follows the OCI standard.

P2P transmission. Wharf [36] and Shifter [20] propose a client-side solution to share local image cache. Dragonfly [14] uses the P2P approach to accelerate deployment or help reduce registry load. Differently, Quicklayer focuses on edge features, using lightweight layer-stack caches and applying P2P to geo-nearby edge clouds.

6 DISCUSSION AND FUTURE WORK

Robustness. In the implementation of Quicklayer, we address several challenges related to distributed storage and message synchronization, drawing inspiration from K8s mechanisms such as asynchronous processing and level trigger, and utilizing file locks (using fcntl( ) interface) and timestamps to ensure cooperative deployment and cache optimization, while also using Linux TC tool to isolate the bandwidth from major services running in edge clouds.

Update optimization. In this preliminary work, we primarily discuss the acceleration of container deployment through container image refactoring. Furthermore, in edge scenarios, frequent configuration changes occur more often, and container version updates are common [17]. This provides significant potential for optimization through refactoring, as many of the files remain unchanged during such updates. In future work, we will explore the benefits of refactoring for updates.

Pull optimization. In the native containerd implementation, the extraction of layers is deferred until all layers are downloaded, and this extraction process is sequential. Currently, despite Quicklayer significantly accelerating the download of required image layers, optimization of the extraction phase has been overlooked. In future work, we will discuss optimization strategies for layer extraction and analyze the benefits they bring. This will make the entire set of acceleration and deployment solutions for container image layers more comprehensive.

7 CONCLUSION

In this preliminary work, we present Quicklayer, a layer-stack-oriented accelerating middleware for fast container deployment in edge clouds, which includes: (1) a layer-based container refactoring solution, (2) a customised K8s scheduler, and (3) distributed shared layer-stack caches. Preliminary results show that Quicklayer can reduce the redundant size of images by up to 3.11× and speed up the deployment process by up to 1.64× compared to the current popular container deployment system.

ACKNOWLEDGMENTS

We would like to express our gratitude to the reviewers for their valuable comments, as well as extend our thanks to PPIO Edge Clouds Co., Ltd., China (www.ppio.cn) for providing the valuable production dataset. This research has received support from the National Science Foundation of China under Grant No. 62072332, the China NSFC (Youth) through Grant No. 62002260, and the Tianjin Xinchuang Haihe Lab under Grant No. 22HHXCJC00002.

REFERENCES

[1] 2009. Apache Mesos. https://github.com/apache/mesos.
[2] 2013. Docker. https://www.docker.com/
[3] 2014. DockerHub. https://hub.docker.com/
[4] 2014. Kubernetes. https://github.com/kubernetes/kubernetes
[5] 2014. Swarm mode overview. https://docs.docker.com/engine/swarm/.
[6] 2015. Cloud native computing foundation. https://cncf.io
[7] 2015. OCI. https://github.com/opencontainers
[8] 2019. containd. https://containerd.io/
[9] 2019. Garbagecollection. https://kubernetes.io/docs/concepts/architecture/garbage-collection/
[10] 2019. GoogleGKE. https://cloud.google.com/kubernetes-engine
[11] 2021. Amazon EKS. https://aws.amazon.com/eks
[12] 2021. Heating problem. https://oracle.github.io/weblogic-kubernetes-operator/faq/node-heating/
[13] 2022. AKS. https://learn.microsoft.com/en-us/azure/aks/
[14] 2023. Dragonfly. https://d7y.io/docs/
[15] Werner Almesberger. 1998. *Linux traffic control-implementation overview*. Technical Report.
[16] 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.

[17] Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara. 2022. Starlight: Fast container provisioning on the edge and over the WAN. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 35–50.

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

[19] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. 2020. Fast and Efficient Container Startup at the Edge via Dependency Scheduling.. In *HotEdge*.

[20] Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. 2017. Shifter: Containers for hpc. In *Journal of physics: Conference series*, Vol. 898. IOP Publishing, 082021.

[21] David Haja, Mark Szalay, Balazs Sonkoly, Gergely Pongracz, and Laszlo Toka. 2019. Sharpening kubernetes for the edge. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*. 136–137.

[22] 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.

[23] Fadia Hasnaoui, Lamia Zohra Mihoubi, Maria Pateraki, and Miloud Bagaa. 2022. Relay-based Network Architectures for Collaborative Virtual Reality Applications. In *GLOBECOM 2022-2022 IEEE Global Communications Conference*. IEEE, 6146–6151.

[24] Billy Kihei, Tyler Davison, Mfon Okpok, and Jim Song. 2022. Comparison of V2N STUN/TURN Round Trip Time Performance on a Public 5G Network. In *2022 IEEE 96th Vehicular Technology Conference (VTC2022-Fall)*. IEEE, 1–5.

[25] Phu Lai, Qiang He, Guangming Cui, Feifei Chen, Mohamed Abdelrazek, John Grundy, John Hosking, and Yun Yang. 2020. Quality of experience-aware user allocation in edge computing systems: A potential game. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 223–233.

[26] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. DADI: Block-level image service for agile and elastic application deployment. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 727–740.

[27] 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*. 3358–3366.

[28] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache.. In *Fast*, Vol. 3. 115–130.

[29] Blesson Varghese, Eyal De Lara, Aaron Yi Ding, Cheol-Ho Hong, Flavio Bonomi, Schahram Dustdar, Paul Harvey, Peter Hewkin, Weisong Shi, Mark Thiele, et al. 2021. Revisiting the arguments for edge computing research. *IEEE Internet Computing* 25, 5 (2021), 36–42.

[30] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.

[31] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.

[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 *USENIX Annual Technical Conference (ATC'20)*.

[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, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. 2019. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–10.

[35] 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.

[36] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S Warke, and Dean Hildebrand. 2018. Wharf: Sharing docker images in a distributed file system. In *Proceedings of the ACM Symposium on Cloud Computing*. 174–185.