PEERSYNC: Accelerating Containerized Model Inference at the Network Edge

Yinuo Deng, Hailiang Zhao, *Member, IEEE*, Dongjing Wang, *Member, IEEE*, Peng Chen, Wenzhuo Qian, Jianwei Yin, Schahram Dustdar, *Fellow, IEEE*, and Shuiguang Deng, *Senior Member, IEEE*

***Abstract*—Efficient container image distribution is crucial for enabling machine learning inference at the network edge, where resource limitations and dynamic network conditions create significant challenges. In this paper, we present PEERSYNC, a decentralized P2P-based system designed to optimize image distribution in edge environments. PEERSYNC employs a popularity- and network-aware download engine that dynamically adapts to content popularity and real-time network conditions. PEERSYNC further integrates automated tracker election for rapid peer discovery and dynamic cache management for efficient storage utilization. We implement PEERSYNC with 8000+ lines of Rust code and test its performance extensively on both large-scale Docker-based emulations and physical edge devices. Experimental results show that PEERSYNC delivers a remarkable speed increase of 2.72×, 1.79×, and 1.28× compared to the Baseline solution, Dragonfly, and Kraken, respectively, while significantly reducing cross-network traffic by 90.72% under congested and varying network conditions.**

***Index Terms*—Container image distribution, edge computing, local area network, model inference, P2P architecture.**

I. INTRODUCTION

MODEL inference at the network edge is becoming increasingly popular in applications such as edge-based video analytics, real-time inference in smart cities, and IoT-based predictive maintenance. To support these scenarios, containerization technologies, such as Docker, are widely used to enable the rapid deployment and management of workloads at scale [1], [2]. Container images, representing multi-layered filesystems, are not merely software artifacts but structured data that must be efficiently stored, distributed, and retrieved across diverse environments, ranging from cloud platforms to the network edge [3], [4], [5], [6], [7], [8].

Deploying a containerized application involves retrieving the necessary image layers from a container registry (e.g., Docker Hub), unpacking them, mounting them using a layered filesystem, and initiating the container entrypoint. These images, structured as a stack of layers representing filesystem segments, are stored and managed in centralized container registries. While these registries efficiently facilitate container image distribution in cloud environments, they encounter unique challenges in edge computing scenarios: (i) *High latency:* Distributed edge devices face substantial delays when fetching image layers over wide-area networks, leading to degraded responsiveness and increased cold-startup times for services [4], [9], [10]; (ii) *Inefficiency under high concurrency:* The non-parallel architecture of container registry uplinks often causes severe bottlenecks when handling concurrent image-pulling requests from multiple edge devices [10]. The growing size of AI models exacerbates this issue. To address these limitations, peer-to-peer (P2P) architectures have been explored to accelerate container image distribution while alleviating the load on centralized registries [2], [4], [7], [8], [11], [12], [13], [14], [15], [16]. P2P systems inherently offer distributed replication, parallel data transfers, and resilience against single points of failure. Although P2P-based solutions have shown promise in cloud environments, their effectiveness in edge settings is constrained by the unique characteristics of edge computing:

- *Limited network bandwidth:* Unlike high-capacity data center networks, edge environments often has significantly constrained bandwidth, hindering fast data exchange [3].
- *Dynamic network topology:* The dynamic nature of edge networks introduces frequent connectivity changes, requiring robust mechanisms to adaptively manage edge devices and maintain consistency without relying on centralized components such as static trackers [17].
- *Low storage scalability:* Portable edge devices, which typically rely on constrained storage media like SD cards or embedded flash memory, often have limited storage. Unlike cloud data centers equipped with technologies like Storage Area Networks [18], their scalability is inherently restricted. Over time, the accumulation of large AI-related

Received 4 April 2025; revised 7 November 2025; accepted 18 December 2025. Date of publication 26 December 2025; date of current version 10 April 2026. This work was supported in part by the National Science Foundation of China under Grant 62125206, Grant 62502441, and Grant 62202131 and in part by the Zhejiang Provincial Natural Science Foundation of China under Grant LD24F020014, Grant LD25F020002, and Grant LZ25F020010. The work of Hailiang Zhao was supported by the Zhejiang University Education Foundation Qizhen Scholar Foundation. *(Yinuo Deng and Hailiang Zhao contributed equally to this work.) (Corresponding authors: Hailiang Zhao; Shuiguang Deng.)*

Yinuo Deng, Peng Chen, Wenzhuo Qian, and Jianwei Yin are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: yinuo@zju.edu.cn; pgchen@zju.edu.cn; qwz@zju.edu.cn; zjuyjw@zju.edu.cn).

Shuiguang Deng is with the First Affiliated Hospital, Zhejiang University School of Medicine, Hangzhou 310003, China, and also with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: dengsg@zju.edu.cn).

Hailiang Zhao is with the First Affiliated Hospital, Zhejiang University School of Medicine, Hangzhou 310003, China, and also with the School of Software Technology, Zhejiang University, Ningbo 315100, China (e-mail: hliangzhao@zju.edu.cn).

Dongjing Wang is with the College of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou 310005, China (e-mail: dongjing.wang@hdu.edu.cn).

Schahram Dustdar is with Distributed Systems Group, TU Wien, 1040 Vienna, Austria, and also with ICREA, UPF, Barcelona, 08002 Barcelona, Spain (e-mail: dustdar@dsg.tuwien.ac.at).

Digital Object Identifier 10.1109/TSC.2025.3648591

TABLE I
COMPARISON BETWEEN PEERSYNC AND STATE-OF-THE-ART WORKS. 'F.D.': 'FULLY DECENTRALIZED', 'S.D.': 'SEMI DECENTRALIZED'.

Work	PEERSYNC	Kraken [16]	Dragonfly [7]	Starlight [10]	EdgePier [9]	FID [8]
Architecture	P2P (F.D.)	P2P (S.D.)	P2P (S.D.)	C/S	P2P (F.D.)	P2P (S.D.)
P2P flavor	Impl. from scratch	Impl. from scratch	Impl. from scratch	-	IPFS	BitTorrent
Scenario	General	Cloud	Cloud	WAN	Edge	General
Tracker placement	dynamic	static	static	-	static	static

images and other data, such as raw sensor outputs, can quickly deplete available storage space.

While P2P-based solutions such as EdgePier [9], Kraken [16], and Starlight [10] address some of these issues and achieve great performance in most cases, they are still based on the traditional P2P architecture and consequently fall short in effectively balancing network efficiency, storage optimization, and adaptability to edge-specific conditions. More specifically, they lack fine-grained control mechanisms and rely on static trackers, making them unsuitable for fluctuating network conditions. Starlight [10] minimizes container startup latency by redesigning deployment protocols and storage formats but requires intrusive modifications to cloud registries, limiting its applicability. Moreover, existing solutions often neglect factors such as dynamic network quality, content popularity, and effective storage utilization, leaving room for improvement. For instance, our preliminary experiments (detailed in Section II) demonstrate that maintaining a single copy of any image layer within a local area network (LAN) can significantly enhance image-fetching speed without overloading local storage.

Crucially, the edge is not a minimized data center. Systems like Kraken [16] and Dragonfly [7], though highly optimized for homogeneous, stable cloud networks, assume persistent centralized trackers or super-nodes that become single points of failure under edge churn. EdgePier [9] improves decentralization but still presumes stable anchor nodes, which rarely exist in mobile or intermittently connected edge deployments. Meanwhile, proposals leveraging IPFS or BitTorrent [8] inherit protocol overheads ill-suited for low-resource devices. Even more recent learning-based approaches [19], [20], [21], [22], while theoretically appealing, require GPU/NPU resources, extensive training data, and stable feedback loops, rendering them impractical on typical edge hardware such as Raspberry Pi. To overcome these challenges, we propose PEERSYNC, a non-intrusive, P2P-based system tailored to the unique demands of the edge. PeerSync constructs a fully decentralized P2P network across different LANs. It features a P2P image download engine that optimizes distribution by periodically calculating the required content pieces for each missing layer of a requested image, leveraging a scoring system that evaluates both content popularity and the quality of the network connection between peers and the requesting edge device. Unlike traditional P2P architectures that rely on manually configured trackers, PeerSync autonomously elects trackers based on real-time metrics, including bandwidth availability and resource utilization, ensuring resilience and performance. PeerSync also employs a selective deletion mechanism to optimize storage utilization, maintaining only essential image layers within each LAN. We compare PeerSync with state-of-the-art solutions in Table I. In summary, our main contributions are as follows:

1) We design an edge container image distribution system PEERSYNC, featuring a high-performance downloading engine, fault-tolerant tracker with self-healing ability, and collaborative dynamic space reclamation for efficient caching.
2) PEERSYNC dynamically assigns scores to peers. Its scoring function jointly optimizes for (i) local network proximity (e.g., same subnet), (ii) real-time bandwidth stability, and (iii) layer popularity. This ensures that the majority of traffic remains within the LAN, directly addressing the high-latency and limited-bandwidth constraints of WAN links.
3) We implement PeerSync with 8000+ lines of Rust code, resulting in a statically linked binary of just 8.8 MB, making it easily deployable on edge devices. PEERSYNC is compatible with the Open Container Initiative (OCI) standard and integrates transparently with existing container runtimes.
4) We conducted extensive experiments to verify PEERSYNC's performance under different network conditions. On average, PEERSYNC achieves 2.72× faster distribution than the Baseline, 1.79× faster than Dragonfly, and 1.28× faster than Kraken. Additionally, PEERSYNC reduces peak cross-network traffic by 90.72% under congested and variable network conditions.

II. MOTIVATION

In this section, we empirically and analytically dissect the limitations of two representative approaches: Kraken [16] and Starlight [10], motivating the need for a more efficient solution tailored to edge computing environments.

A. Traditional P2P Approaches

P2P architectures such as Napster [23], BitTorrent [24], and IPFS [25] enable decentralized content sharing by allowing nodes to act as both consumers and providers. While these systems excel in cloud or data center settings with abundant bandwidth and stable connectivity, they suffer from a critical flaw in edge environments: *the lack of bandwidth awareness and network topology sensitivity*. This leads to inefficient utilization of scarce inter-LAN links and undermines the very benefits P2P promises.

To illustrate this, we conducted an experiment using Kraken [16], which builds upon BitTorrent's protocol. As shown on the left side of Fig. 1, our testbed comprised two LANs connected via a 100 Mbps link, representative of typical edge-to-edge or edge-to-cloud uplinks. Each LAN included three Raspberry Pi 4 Model B devices (each with 4 GB RAM and 32 GB eMMC storage) connected through a 1 Gbps switch.

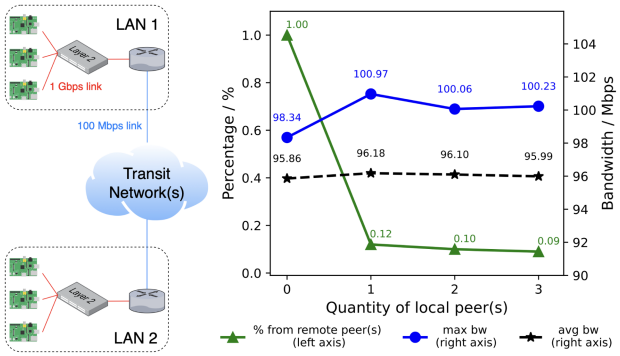

Fig. 1. An edge computing environment with 2 LANs and the corresponding observed results when using BitTorrent for image downloading.

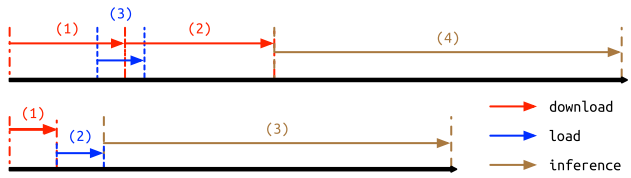

Fig. 2. Timeline diagram of a large model inference task. *Top: Lazy Loading (e.g., Starlight [10]):* Download components (PyTorch, cuDNN library, etc.) → Download the model → Load the ML framework → Perform inference. *Bottom: P2P (e.g.,* PEERSYNC*):* Download the full image → Load the ML framework → Perform inference.

In LAN 1, two hosts served as seeders for a large container image; download requests originated from LAN 2, with 1, 2, and 3 local peers progressively added as potential sources. The results (right side of Fig. 1) reveal a striking inefficiency: even when local peers were available in LAN 2, Kraken still fetched approximately 10% of the image blocks from remote peers in LAN 1. Crucially, this small fraction consumed over 95% of the inter-LAN bandwidth. This behavior stems from BitTorrent's peer selection strategy, which prioritizes global peer diversity and piece availability over network locality. As a result, the narrow uplink becomes saturated, delaying downloads for all nodes and negating the latency advantage of local caching. Beyond bandwidth waste, P2P systems like Kraken impose significant storage pressure on edge devices. In a separate experiment, we distributed the top-10 most popular Docker Hub images across Kraken nodes. After downloading and decompressing these images, each node consumed 1408.54 MiB of disk space. For resource-constrained edge devices (e.g., those using SD cards or embedded flash with limited write endurance), such storage overhead is unsustainable, especially when images accumulate over time alongside sensor data or model checkpoints.

Furthermore, Kraken's reliance on a centralized tracker for peer discovery introduces a single point of failure. In dynamic edge environments, where devices frequently join, leave, or experience intermittent connectivity, tracker outages lead to network fragmentation, prolonged peer discovery latency, and often require manual intervention to restore service [26], [27], [28]. This centralized coordination model is fundamentally at odds with the autonomy and resilience required in edge deployments.

B. Image Structure-Based Approaches

An alternative strategy, exemplified by Starlight [10], seeks to minimize container startup latency by analyzing image structures and runtime behavior. Starlight removes unnecessary components (a process known as debloating [29], [30]) and decouples the download and execution phases through lazy loading: it prioritizes "hot" files (e.g., libraries) based on runtime traces collected during a profiling run. While effective for certain applications, this approach faces three critical limitations in edge AI scenarios:

- *Starlight assumes workload homogeneity and predictability:* Starlight requires users to convert images into a custom format and execute them once to collect file-access traces. This profiling step assumes that future executions will follow similar patterns, which breaks down when deploying diverse or evolving AI workloads (e.g., switching from object detection to LLM inference). At scale, maintaining accurate traces for hundreds of edge applications becomes impractical.
- *Starlight offers minimal benefit for large-model inference tasks:* As shown in Fig. 2, inference pipelines are typically dominated by two phases: (1) fetching the model and framework, and (2) computation. Crucially, the entire model must be available before inference can begin, as models are often stored as a few large files (e.g., in Safetensors [31] or PyTorch format [32]). Consider Meta's Llama 3.1 [33] (`ai/meta-llama:3.1-8B-Instruct-cuda-12.6`). When uncompressed, this image occupies approximately 21.4 GiB of disk space, composed of only two major parts: *Model weights:* 15 GiB (70.10%), stored in just 4 files using Safetensors [31]; *ML framework and dependencies:* 6.2 GiB (28.97%), including PyTorch [32] and CUDA libraries. In such cases, downloading libraries first provides negligible speedup since the critical path remains the transfer of the 15 GiB model file. Trace-based methods like Starlight cannot accelerate this bottleneck because they optimize *what* and *when* to download, not *how fast* the data can be fetched.

In addition, Starlight remains dependent on centralized registries. Despite its optimizations, it still pulls base images from Docker Hub or private registries, inheriting all the latency, bandwidth, and availability challenges of cloud-centric architectures. In disconnected or bandwidth-constrained edge settings, this dependency severely limits deployability.

C. Insights

Both Kraken and Starlight illustrate shortcomings in existing approaches. Kraken, while leveraging P2P architectures, suffers from bandwidth inefficiencies and excessive reliance on centralized trackers. Starlight, despite its innovative lazy-loading approach, struggles with scalability across diverse model inference workloads and fails to address large model file distribution effectively. These limitations highlight the need for a decentralized, bandwidth-aware solution that maximizes local resource utilization, minimizes bandwidth waste, and adapts dynamically to the constraints of edge environments.

III. THE PEERSYNC SYSTEM

A. Problem Formulation

We consider a distributed environment where nodes (including cloud instances and edge devices) pull container images from a central registry. Each image consists of a manifest and multiple immutable layers. When many nodes concurrently pull the same image, the registry becomes a bottleneck, leading to

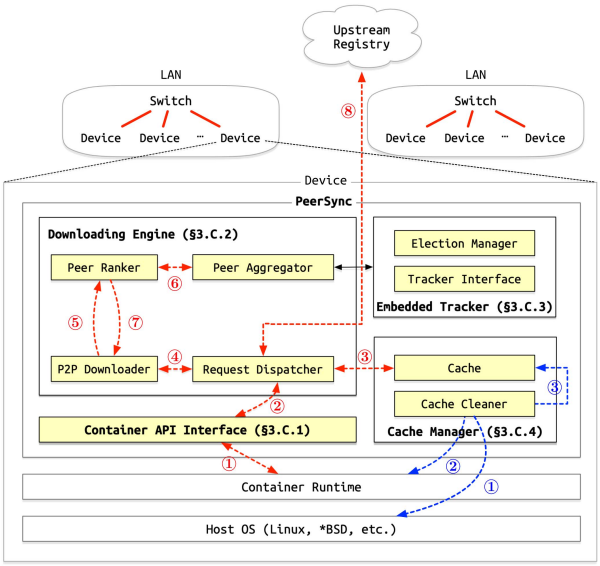

Fig. 3. The architecture of PEERSYNC.

high latency and WAN bandwidth consumption. Let $\mathcal{P}$ be the set of peers, each holding a subset of layers $\mathcal{L}_p \subset \mathcal{L}$, where $p \in \mathcal{P}$. Peers are connected via heterogeneous, dynamic network links with unknown bandwidth and availability. A requesting peer p_{req} must fetch all layers of a target image as quickly as possible, without prior knowledge of which peers have which layers or their current upload capacity. The problem is to minimize the end-to-end image pull time by opportunistically leveraging P2P transfers among peers, while operating without global coordination, runtime modifications, or assumptions about peer stability or connectivity.

B. System Architecture Overview

PEERSYNC is designed as a drop-in enhancement for existing container ecosystems. As illustrated in Fig. 3, the system comprises four core components: *API Interface*, *Downloading Engine*, *Embedded Tracker*, and *Cache Manager*, which are orchestrated along two orthogonal threads: (i) the pull path, which handles on-demand layer retrieval, and (ii) the cache maintenance path, which runs continuously in the background to manage storage and peer metadata. The entire system exposes a standard OCI Distribution API, making it fully compatible with Docker, Podman, and other OCI-compliant clients. No modifications to the container runtime or user workflow are required; peers simply configure their client to use PEERSYNC as the registry endpoint.

A typical pull request begins when a container runtime issues an HTTP GET for a layer. The request is intercepted by the *API Interface*, which checks local cache availability. If the layer is absent, the request is forwarded to the *Downloading Engine*, where a lightweight *Request Dispatcher* decides whether to fetch the layer from upstream (via HTTP) or initiate a P2P transfer based on multiple factors. When P2P is selected, the engine queries the *Embedded Tracker* for candidate peers and coordinates block-level downloads. Upon successful retrieval, the layer is stored in the local *Cache*, and the response is returned to the client. Concurrently, the *Cache Manager* monitors global access patterns across layers and performs eviction based on a cost-aware policy that considers layer size, replication count within the LAN, and reuse likelihood. The *Embedded Tracker*, operating independently, maintains an up-to-date view of nearby peers through periodic gossip and self-election, ensuring coordination remains decentralized and resilient.

PEERSYNC attempts to join the P2P swarm during application startup. During bootstrap, PEERSYNC reads predefined nodes from the configuration and broadcasts bootstrap messages locally to join the swarm. If the bootstrap phase fails, PEERSYNC runs in a degraded mode as a proxy for the upstream registry and continues to bootstrap until a full P2P connection can be established. The predefined node list is dynamically updated with high-uptime peers, enabling PEERSYNC to withstand environmental changes.

C. Component Design

1) Container API Interface: Modern container ecosystems achieve remarkable interoperability through adherence to the OCI standards. The OCI defines both an *Image Specification* (governing image format and manifest structure) and a *Distribution Specification* (defining HTTP-based registry APIs for pulling images). To ensure PEERSYNC operates as a drop-in replacement without requiring any modification to existing runtimes or user workflows, it must fully comply with these standards. Crucially, the container runtime initiates image pulls via a well-defined sequence: one request for the image manifest (a JSON document listing layer digests and metadata), followed by one request per missing layer (each returning a gzipped tarball). Any deviation from this protocol would break compatibility. Thus, the primary motivation is to provide a transparent, standards-compliant facade that hides the internal P2P complexity while preserving the exact API contract expected by the runtime.

The Container API Interface implements a subset of the OCI Distribution Specification. It serves as the sole external entry point for the container runtime.

- *Manifest Handling.* The manifest is typically small (often < 5 KB) but frequently accessed and subject to upstream updates (e.g., LATEST tag). To ensure low-latency responses and immediate consistency, manifests are stored in an in-memory cache. This cache is kept up-to-date by periodically polling the upstream registry or by invalidating entries upon detecting a new pull request for a potentially stale tag.
- *Layer Handling.* Layer requests are not served directly. Instead, the interface acts as a lightweight dispatcher, forwarding each blob request (identified by its digest) to the Downloading Engine (Section III-C). The engine is responsible for fulfilling the request, whether from local cache, a LAN peer, or the upstream registry, and streaming the data back through this interface.

2) Downloading Engine: A naive P2P approach, which attempts to find peers for every layer, fails to account for the empirical reality of container image composition. As shown in Table II, nearly half of all layers are smaller than 512 KiB, with a median size of just 1.03 MiB. For such small payloads, the latency of peer discovery in a Distributed Hash Table (DHT) or even a local multicast can exceed the time required to download the layer directly from a nearby registry. Conversely, large layers (e.g., base OS images) can benefit immensely from parallel, multi-source P2P downloads. The engine's core motivation is to make a per-layer, context-aware decision that optimizes for both

TABLE II
LAYER SIZE DISTRIBUTION OF DOCKER HUB TOP 100 IMAGES. EACH PERCENTAGE REPRESENTS THE PROPORTION OF LAYERS THAT ARE SMALLER THAN THE SPECIFIED THRESHOLD.

Thres. (B)	% < Thres.	Thres. (B)	% < Thres.
128	1.64	1 Ki	29.27
8 Ki	41.45	512 Ki	47.78
4 Mi	57.38	32 Mi	76.81
256 Mi	97.19	605.73 Mi	100.00

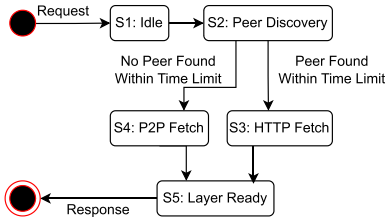

Fig. 4. The state machine of P2P downloading in PEERSYNC.

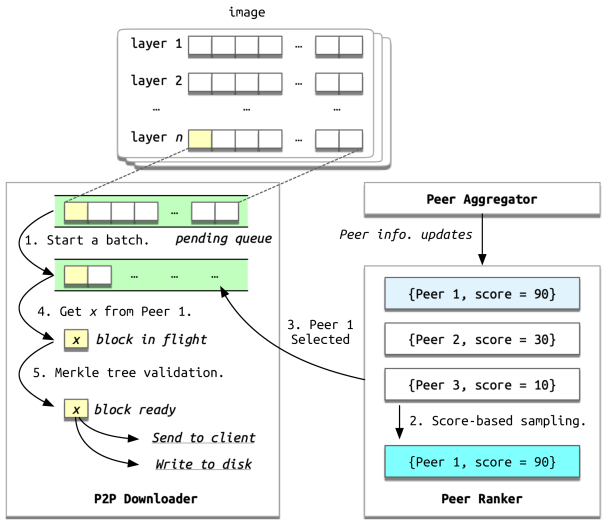

Fig. 5. Workflow of P2P downloading in PEERSYNC.

speed and system stability, dynamically choosing between P2P and direct upstream fetching.

The Downloading Engine is structured around a *Request Dispatcher* and a *P2P Downloader*.

a) Request Dispatcher: Upon receiving a layer digest from the API interface, the dispatcher first checks with the local Cache Manager (Section III-C4) and serves the content if locally present. Otherwise, a three-stage decision process is executed to retrieve the content from external sources. (i) It first queries the other nodes in the same LAN and fetches from the neighbor nodes. (ii) If not local, the dispatcher consults the manifest cache to obtain the layer's size L_i to determine if a P2P process shall be started. If L_i is below a configurable threshold θ (e.g., 1 MiB), the request is routed directly to the upstream registry. This is justified by the data in Table II, which shows that a policy of direct fetch for small layers can handle a large fraction of requests with minimal latency. (iii) For layers larger than θ, the dispatcher initiates a P2P discovery process. However, this process is bounded by a dynamic timeout τ, defined as the 95th percentile of recent round-trip times (RTTs) over a 10-second sliding window. If no suitable peer is found within τ, the system falls back to the upstream registry to avoid indefinite user wait times. This logic can be viewed as a state machine (Fig. 4), but is now implemented as a streamlined, asynchronous workflow to reduce overhead.

b) Popularity- and Network-Aware P2P Downloader: PEERSYNC segments each image layer into fixed-size *blocks* to enable concurrent downloads from multiple peers. Upon receiving a download request from the Request Dispatcher, the P2P Downloader initiates a five-stage workflow (Fig. 5): (i) select a batch of blocks for the current cycle; (ii) sample candidate peers based on dynamically updated scores maintained by the Peer Aggregator; (iii) assign each block to the highest-scoring peer; (iv) issue download requests; and (v) validate received blocks via cryptographic hash. During this phase, PEERSYNC re-queues failed blocks or caching and delivers valid ones.

Determining the block size. Block size critically balances parallelism and overhead. Excessively large blocks limit concurrency; overly small ones inflate Merkle tree depth and hash computation costs. Empirical evaluation using an 8194.5 MiB image in a 10 Gbps LAN (Table III) shows optimal performance at 16-128 MiB. Accordingly, PEERSYNC sets block size L_b adaptively based on image size L_i:

TABLE III
DOWNLOAD TIMES FOR AN 8194.5 MIB IMAGE IN A 10 GBPS LAN ENVIRONMENT WITH DIFFERENT BLOCK SIZES

Block Size (MiB)	#Blocks	Download Time (s)
256	33	58
128	65	59
32	257	64
16	513	63
8	1025	87

$$L_b = \begin{cases} L_i/256 & \text{if } L_i \geq 1024 \text{ MiB}, \\ L_i/64 & \text{if } 256 \text{ MiB} \leq L_i < 1024 \text{ MiB}, \\ L_i/16 & \text{if } 16 \text{ MiB} \leq L_i < 256 \text{ MiB}, \\ L_i & \text{otherwise.} \end{cases} \quad (1)$$

For tiny images whose size $L_i < 16$ MiB, a single block is used to minimize coordination overhead.

Peer selection rules. Peer selection combines three criteria into a unified utility score $U(p;t) \in [0, 100]$: network position content popularity, and customized logic.

1) *Network position.* Peer selection uses a *network-aware* score that favors peers with higher effective throughput. LAN-local peers receive the maximum score of 100. For remote peers, the score is based on observed download speed, not low-level metrics like packet loss, as congestion control is handled by the OS kernel [34], [35], [36], [37]; using throughput avoids interfering with kernel mechanisms and better reflects end-to-end performance. Each peer p maintains a sliding window $\mathcal{W}_p$ of past speeds $\{s_p^{t'}\}$. The current speed estimate s_p^t is an exponentially weighted average that prioritizes recent samples:

$$s_p^t = \frac{\sum_{s_p^{t'} \in \mathcal{W}_p} s_p^{t'} \cdot e^{L-t'}}{\sum_{s_p^{t'} \in \mathcal{W}_p} e^{L-t'}}, \quad (2)$$

where L is the current logical time. A global baseline $\bar{s}^t$ is computed similarly over all recent transfers:

$$\bar{s}^t = \frac{\sum_{\bar{s}^{t'} \in \mathcal{W}} \bar{s}^{t'} \cdot e^{L-t'}}{\sum_{\bar{s}^{t'} \in \mathcal{W}} e^{L-t'}}. \tag{3}$$

The raw network advantage is $\text{net}_p^t = s_p^t - \bar{s}^t$, which is then linearly rescaled to [0,100] (with negatives clamped to the minimum known score) to yield the final network score. This relative scoring adapts to heterogeneous and dynamic network conditions.

2) *Content popularity.* Content popularity discourages reliance on peers holding rare layers. Let $\rho_l \in [0,1]$ denote the fraction of known image instances containing layer l:

$$\rho_l = \frac{\sum_{p \in \mathcal{P}^t} \sum_{i \in \mathcal{I}_p^t} \epsilon_l^i}{\sum_{p \in \mathcal{P}^t} \sum_{i \in \mathcal{I}_p^t} 1}, \quad \epsilon_l^i = \begin{cases} 1 & \text{if } l \in i, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Peers periodically exchange layer sets via differential updates; unchanged states are acknowledged with sequence numbers to minimize overhead. The popularity score for peer p is then:

$$\text{pop}_p^t = 100 \times \left(1 - \frac{\sum_{i \in \mathcal{I}_p^t} \sum_{l \in \mathcal{L}_i} e^{-\lambda \cdot \rho_l}}{\sum_{i \in \mathcal{I}_p^t} \sum_{l \in \mathcal{L}_i} 1}\right), \tag{5}$$

which down-weights peers storing rarer content. The rationale is to preserve the bandwidth of these critical peers, ensuring they remain available to serve the rare content that only they possess, thus enhancing the overall availability of all layers in the swarm.

3) *Extensibility via custom scoring.* To accommodate the diverse and heterogeneous nature of edge deployments, PEERSYNC's scoring function is designed with an extensibility hook for domain-specific policies, represented by a custom score cst_p^t. This architectural choice allows administrators to inject logic tailored to their unique operational constraints without modifying the core system. For instance, in a wireless edge environment, an administrator may wish to deprioritize peers with weak signal strength to conserve their battery and bandwidth. This can be achieved by defining the custom score cst_p^t as a function of the peer's real-time signal strength, thereby penalizing nodes with poor connectivity in the selection process.

The total utility is a weighted sum:

$$U(p;t) = \alpha \cdot \text{net}_p^t + \beta \cdot \text{pop}_p^t + \gamma \cdot \text{cst}_p^t, \tag{6}$$

with $\alpha + \beta + \gamma = 1$. In the default configuration, $\alpha = \beta = 0.5$, and custom scoring is disabled. The three weights may be adjusted accordingly to achieve a balance between network optimization and resilience. A high α value indicates a strong preference for nearby peers, while β strengthens availability. γ is configured along with the custom score for more flexibility. Eventually, peer selection uses a softmax sampling over the utility scores $U(p;t)$, favoring high-utility peers while preserving space for exploration.

3) Embedded Autonomous Tracker: In P2P networks, trackers play a central role in accelerating peer discovery [17]. Without an active tracker, nodes must fall back to multi-hop DHT lookups, which significantly increase discovery latency and degrade download performance. However, traditional trackers introduce a single point of failure, vulnerable to crashes, network partitions, or misconfiguration, and require separate deployment and maintenance.

To eliminate this dependency, PEERSYNC embeds an autonomous tracker module that self-activates when no live trackers are reachable. Specifically, if a node fails to contact any known tracker during bootstrap or periodic health checks, it initiates a leader election within its local network partition using the FLOODMAX algorithm [38]. In this protocol, each candidate broadcasts a stability score (based on metrics including uptime and known neighbors) and propagates the highest score observed so far. The node with the globally maximal score becomes the new tracker. While FLOODMAX ensures correctness, its naive implementation incurs $O(n^2)$ message complexity in dense networks. To address this, PeerSync applies path-pruning techniques [39] that suppress redundant message forwarding across already-explored paths and avoid cross-subnet flooding unless necessary. This reduces communication overhead to near-linear complexity while preserving convergence guarantees.

Since the tracker logic is fully integrated into PEERSYNC's runtime, no external configuration or manual intervention is required. This design not only removes operational overhead but also enhances system resilience, which can ensure continuous peer discovery even in transiently partitioned or infrastructure-free edge environments.

4) Cache Manager: Efficient storage management is critical for low-end edge devices, which typically operate under severe resource constraints and rely on flash-based storage without advanced virtualization layers like Ceph [40], [41]. Over time, PEERSYNC accumulates cached images, even after their associated applications have terminated, consuming disk space needed for data-intensive tasks such as storing sensor logs or datasets. To preserve system stability while maintaining performance, PEERSYNC must minimize its storage footprint without compromising image availability. To this end, PEERSYNC implements a dynamic cache cleaner that monitors image usage via the container runtime and triggers eviction when free space falls below a threshold (e.g., 10% or a user-defined limit). Eviction decisions are guided by three factors: (i) last access time, (ii) image size, and (iii) content popularity, considering both local presence and global replication.

The underlying algorithm extends the classic Least Recently Used (LRU) policy by incorporating *cache miss cost*, which reflects the non-uniform cost of retrieving an image after eviction [42], [43]. Unlike RAM-based caching, in which access latency is uniform, image retrieval in PEERSYNC varies significantly by network context:

- Images with multiple local replicas (within the same LAN) incur near-zero miss cost and are safe to evict.
- Images that are the sole local copy are assigned a miss cost inversely proportional to the number of remote replicas.
- Sole known copies are assigned the highest retention priority and evicted only under extreme storage pressure.

By jointly optimizing recency, space, and retrieval cost, this popularity-aware strategy enables PEERSYNC to balance disk utilization and performance on resource-constrained edge nodes.

D. Implementation Concerns

PEERSYNC adopts a message-passing architecture in which each functional component runs as an independent asynchronous task. Components communicate exclusively through well-defined, typed message interfaces, decoupling internal logic from inter-component coordination. A crash or stall in one module (e.g., due to a malformed image manifest or network timeout) does not propagate to others. The supervisor

process can restart the failed component within milliseconds, ensuring continuous operation without full-system disruption. In addition, new peer discovery protocols, storage backends, or scoring heuristics can be integrated by implementing standardized interfaces without modifying core logics. This facilitates rapid adaptation to upcoming container formats or edge-specific constraints.

Peer discovery leverages multiple orthogonal mechanisms, centralized trackers, Kademlia-based DHT, and LAN-local multicast, all operating concurrently. The Downloading Engine monitors availability signals from these sources and initiates block fetching as soon as any peer in the P2P swarm reports possession of the requested layer, mimicking BitTorrent's opportunistic pull model. If no P2P sources are reachable within a configurable timeout (or if the content is globally unique), PEERSYNC transparently falls back to direct HTTP(S) retrieval from the upstream registry. This hybrid strategy guarantees liveness while maximizing bandwidth utilization whenever collaborative sources exist.

IV. Adaptive Peer Selection and System-Level Guarantees

Peer selection in PEERSYNC operates in a non-stationary, partially observable environment where performance and content distribution vary constantly. Rather than pursuing unattainable optimality guarantees against an ill-defined oracle, we analyze the mechanism through three analytically tractable lenses: (i) convergence of utility estimation under exponential smoothing, (ii) probabilistic protection of rare-content peers, and (iii) approximation quality of aggregate throughput maximization. Together, these properties explain why the heuristic consistently achieves high speedup while preserving swarm health.

A. Convergence of Network Utility Estimation

The network score net_p^t is derived from an exponentially weighted moving average (EWMA) of observed download speeds (see (2)), where $\tau_s > 0$ is the smoothing time constant (implicit in your original notation via $L - t'$). Let $B_p(t)$ denote the true instantaneous throughput achievable from peer p at time t, and assume $|s_p^{t'} - B_p(t')| \le \sigma$ for all t' (bounded measurement noise). Then, under mild Lipschitz continuity of $B_p(\cdot)$ (i.e., $|B_p(t) - B_p(t-1)| \le \delta$), the estimation error satisfies

$$|s_p^t - B_p(t)| \le \sigma + \delta \cdot \tau_s. \tag{7}$$

Thus, by choosing τ_s appropriately (e.g., $\tau_s = 2$ rounds in our implementation), the EWMA estimator tracks the true throughput within a bounded error envelope. Since net_p^t is a monotonic rescaling of $s_p^t - \bar{s}^t$, it inherits this stability, ensuring that transient anomalies do not dominate peer ranking.

B. Probabilistic Protection of Rare-Content Peers

Let $\mathcal{R}_t = \{l : \rho_l < \rho_{\min}\}$ denote the set of rare layers at time t, and define the rarity exposure of peer p as

$$\eta_p^t = \frac{1}{|\mathcal{L}_p^t|} \sum_{l \in \mathcal{L}_p^t} \mathbb{I}[l \in \mathcal{R}_t], \tag{8}$$

where $\mathcal{L}_p^t$ is the set of layers held by p and $\mathbb{I}[\cdot]$ is the indicator function. The popularity score can be upper-bounded as

$$\text{pop}_p^t \le 100 \cdot (1 - e^{-\lambda \rho_{\min}}) \cdot (1 - \eta_p^t) + 100 \cdot (1 - e^{-\lambda}) \cdot \eta_p^t. \tag{9}$$

Consequently, if $\eta_p^t \ge \eta_0$ (i.e., peer p holds many rare layers), then $\text{pop}_p^t \le \bar{u}_{\text{rare}} < 100$. Under softmax selection with temperature τ, the probability of selecting such a peer is upper-bounded by

$$\begin{aligned} \Pr\{p_t = p\} &\le \frac{\exp\left((\alpha \cdot 100 + \beta \cdot \bar{u}_{\text{rare}})/\tau\right)}{\exp\left((\alpha + \beta) \cdot 100/\tau\right)} \\ &= \exp\left(-\frac{\beta(100 - \bar{u}_{\text{rare}})}{\tau}\right). \end{aligned} \tag{10}$$

This exponential suppression ensures that peers critical for rare content are selected infrequently, thereby conserving their bandwidth. In steady state, this mechanism bounds the expected depletion rate of rare layers, enhancing long-term availability.

C. Throughput Approximation Via Utility Maximization

Let $\mathcal{S}_t \subseteq \mathcal{P}$ be the set of k peers selected for concurrent download at time t. Assume block-level parallelism and negligible coordination overhead, so the total bandwidth is

$$B_{\text{total}}(t) = B_0 + \sum_{p \in \mathcal{S}_t} B_p(t). \tag{11}$$

Since net_p^t is a linear transformation of s_p^t, and s_p^t approximates $B_p(t)$ within error $\epsilon = \sigma + \delta\tau_s$, we have

$$\left| \sum_{p \in \mathcal{S}_t} \text{net}_p^t - \sum_{p \in \mathcal{S}_t} B_p(t) \right| \le k \cdot C \cdot \epsilon, \tag{12}$$

for some constant C from the rescaling. Because $\text{pop}_p^t \in [0, 100]$ is independent of instantaneous bandwidth, the total utility sum satisfies

$$\sum_{p \in \mathcal{S}_t} U(p;t) = \alpha \sum_{p \in \mathcal{S}_t} \text{net}_p^t + \beta \sum_{p \in \mathcal{S}_t} \text{pop}_p^t + \gamma \sum_{p \in \mathcal{S}_t} \text{cst}_p^t. \tag{13}$$

Thus, maximizing $\sum_{p \in \mathcal{S}_t} U(p;t)$ approximately maximizes $\sum_{p \in \mathcal{S}_t} B_p(t)$ up to an additive error of $O(k\epsilon)$. In practice, since $\alpha = \beta = 0.5$, the scheduler jointly optimizes for speed and resilience, achieving a Pareto-efficient trade-off.

V. Evaluation

We conduct extensive experiments in two environments: a Docker Compose-based emulation on a high-performance x86_64 host (Intel Xeon Silver 4214) and physical edge devices (Raspberry Pi 4 Model B). The emulation enables the simulation of large-scale environments, while the physical setup validates the practical feasibility of PEERSYNC. We select a range of popular AI/ML and commonly used applications of varying sizes, from small base images to large language model (LLM) containers (Table IV). These images represent diverse use cases, allowing us to evaluate system performance under varying workloads.

We evaluated PEERSYNC against two popular container image distribution systems: *Dragonfly* [7] and *Kraken* [16], as well as the plain HTTP-based pull method, referred to as the *Baseline*.

TABLE IV
THE CONTAINER IMAGES CHOSEN FOR EVALUATION

Name	Tag	Service	Compressed Size	Description
redhat/granite-3-1b-a400m-instruct	latest	NLP	1.47 GB	1B finetuned IBM Granite 3.0 [44]
ai/meta-llama	3.1-8B-Instruct	NLP	14.91 GB	Llama 3.1 model by Meta [33]
cvisionai/segment-anything	latest	Vision	5.2 GB	Segment Anything model by Meta [45]
langchain/langchain	latest	NLP	437.57 MB	LLM application adaptation framework [46]
pytorch/pytorch	2.5.1-cuda12.4-cudnn9-runtime	General	3.11 GB	Deep learning framework [32]
tensorflow/tensorflow	nightly-gpu	General	3.61 GB	Deep learning framework [47]

- Dragonfly, hosted by the CNCF as an Incubating Level Project, leverages P2P technology to accelerate content distribution. It is designed to enhance efficiency in distributed environments by reducing reliance on centralized registries.
- Kraken, an open-source project developed by Uber and deployed in production since early 2018, also employs P2P technology for Docker image management, replication, and distribution. Kraken is particularly tailored for hybrid cloud environments, offering decentralized capabilities to optimize large-scale image deployments.

Although Starlight [10] also focuses on improving container image delivery, it relies on fundamentally different mechanisms, such as runtime trace collection and dependence on centralized registries. These differences make direct comparisons with PEERSYNC less relevant (see Section II). Consequently, we did not include Starlight in our comparisons.

A. Docker Compose-Based Emulation

1) Experimental Setup: We deployed a Docker Compose-based emulation environment comprising 10 Linux bridge networks (`net_worker{`n`}`, $n = 1$ to 10) to model a distributed system of interconnected edge sites. Each network hosted one router responsible for inter-network communication and seven worker nodes issuing container image pull requests. Routers were configured using `tc(8)` [48] to impose realistic WAN characteristics, including bandwidth limits (20Mbps to 500Mbps), variable latency, and packet loss, mimicking ISP and transit links. In contrast, intra-network communication within each LAN was provisioned with uncapped bandwidth and zero packet loss to reflect typical local cluster conditions. This multi-LAN environment models the Internet, where LANs are edge sites and routers are ISPs. In each edge site, traffic is routed internally without flowing through external ISPs.

To emulate heterogeneous user demand, we modeled image request arrival times using an inverse Poisson process:

$$t_i \sim \text{Poisson}^{-1}\left(\text{random}\left(0.001, A \cdot e^{\frac{B}{s_i}}\right)\right),$$

where s_i denotes the size of image i, and A, B are tunable parameters controlling request intensity. Larger values of A and B correspond to higher request frequencies, simulating peak-load scenarios. Background traffic generated via `iPerf` [49] introduced additional congestion on shared links, better reflecting real-world multi-tenant network usage. All centralized services, including container registries, Dragonfly metadata databases, and Kraken trackers, were co-located in `net_worker1` to emulate a cloud-hosted control plane.

TABLE V
KEY PARAMETERS USED IN THE EMULATION

Parameter	Value
Number of LANs	10
Number of Nodes per LAN	7
Emulation Timespan	1800 seconds
Bandwidth (Internal)	Uncapped
Packet Loss (Internal)	0%
Bandwidth (Stable Net.)	Uncapped
Packet Loss (Stable Net.)	0%
Bandwidth (Congested Net.)	100 to 500Mbps
Packet Loss (Congested Net.)	0 to 10%
Bandwidth (Varying Net., Min)	20 to 150Mbps
Packet Loss (Varying Net., Min)	0 to 10%
Bandwidth (Varying Net., Max)	100 to 500Mbps
Packet Loss (Varying Net., Max)	30 to 50%
Peer Churn Ratio (Varying Net.)	10%
Environmental Variation Cycle (Varying Net.)	60 seconds
Node CPU Allocation	2 to 4 shared cores
Node RAM Allocation	2 to 4 GiB
PEERSYNC Scoring Weight α	0.5
PEERSYNC Scoring Weight β	0.5

For reproducibility of our evaluation, Table V gives full parameters used in our emulation. These parameters include values for network bandwidth and packet loss, node CPU and RAM allocation, and how we configure PEERSYNC to balance between network and popularity-aware scoring. For the Varying Network experiment group, the bandwidth and packet loss values are randomly modified within the range specified.

2) Image Distribution Time: Fig. 6 presents the average container image distribution times under different network conditions as a function of request frequency (controlled by parameter A). Due to the wide dynamic range, especially under high load, a base-2 logarithmic scale is used on the y-axis for clarity.

a) Stable network conditions (Fig. 6(a)): In environments with sufficient and stable network resources, all P2P-based methods, including PEERSYNC, exhibit similar performance trends as request frequency increases (see Table VI). While distribution latency increases slightly with higher request loads, P2P methods consistently outperform the Baseline HTTP approach. This aligns with the known benefits of P2P models, such as efficient data sharing and reduced reliance on centralized

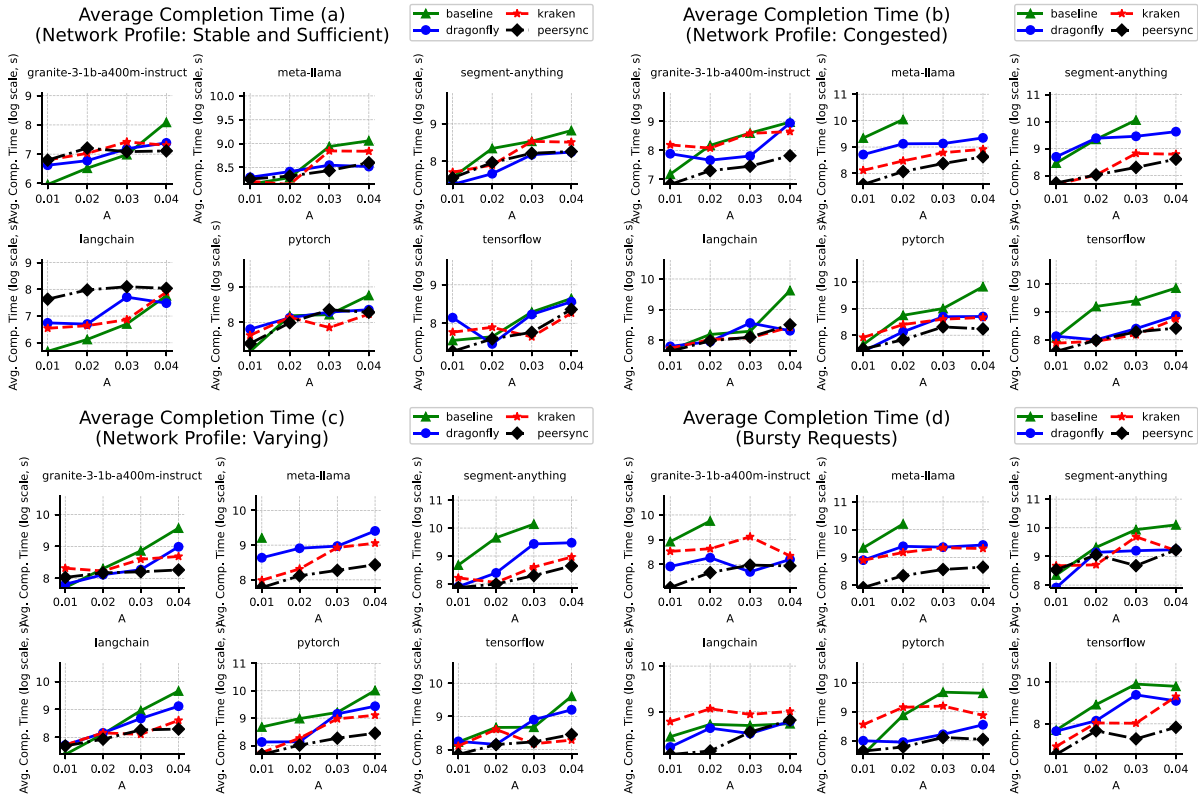

Fig. 6. Average AI/ML image distribution time under various network conditions (log scale). The x-axis is the parameter A. The missing points mean that the download operation could not be completed during the limited time frame.

TABLE VI
NORMALIZED AVERAGE COMPLETION TIME (BASELINE = 100%, LOWER IS BETTER) UNDER DIFFERENT NETWORK PROFILES

Profile	Baseline	Dragonfly [7]	Kraken [16]	PEERSYNC
Stable	100%	75.01%	80.13%	**77.85%**
Congested	100%	64.27%	61.89%	**45.89%**
Varying	100%	65.63%	46.96%	**36.71%**

registries that have been extensively validated in data center contexts. The trend is particularly pronounced for large LLM images: the average distribution time roughly quadruples as request frequency increases, highlighting the critical role of P2P technologies in facilitating the deployment of modern AI/ML models with billions of parameters.

b) Congested network conditions (Fig. 6(b)): In environments with bandwidth limitations, packet loss, and high latency, the Baseline method suffers severe performance degradation due to the long-tail effects of centralized pulling. Remote data transfers are particularly affected, often exceeding the default time limit of 1200 seconds, as indicated by the missing points in the green Baseline lines. In contrast, PEERSYNC and other P2P-based methods maintain better performance by distributing the load across multiple peers, effectively mitigating bandwidth constraints. Notably, Dragonfly shows a sharp performance drop for larger images due to its reliance on centralized components such as schedulers and managers. As communication between edge nodes and these components becomes unreliable, Dragonfly's ability to coordinate cloud-edge interactions weakens, resulting in bottlenecks during image serving. PEERSYNC, by contrast, leverages its decentralized architecture to achieve consistent performance, even under congestion.

c) Variable network conditions (Fig. 6(c)): This scenario simulates real-world edge environments where latency, packet loss, and bandwidth fluctuate significantly, and participating nodes frequently join and leave the network. The corresponding parameters are given in Table V. In each variation cycle, every emulated router adopts a new random configuration within the predefined range, 10% of the currently active peers leave the swarm, and a subset of previously disconnected peers rejoin the swarm. Such instability disrupts inter-peer connections and hinders image fetching, even for smaller images like `langchain/langchain`. Compared to other P2P-based solutions, PEERSYNC demonstrates superior resilience under these conditions. By prioritizing local peer synchronization and minimizing dependence on centralized components, PEERSYNC reduces reliance on specific, potentially unstable links and dynamically explores all available links to maximize performance. This strategy not only improves distribution efficiency but also enhances fault tolerance, which is critical in environments with unpredictable network behavior.

PEERSYNC's ability to optimize local storage and peer communication makes it particularly robust in dynamic edge environments. As shown in Table VI, PEERSYNC outperforms the Baseline, Dragonfly, and Kraken under unstable, high-frequency scenarios by average factors of 2.72, 1.79, and 1.28, respectively.[1] The speedup is even more pronounced for larger

[1]The timeout for each request was 1200 seconds. In some instances, the Baseline could not finish within the limit. We assigned 1200 seconds to the missing data points to avoid missing points. Consequently, the actual speedup relative to the Baseline is higher than the calculated value.

TABLE VII
AGGREGATE INTER-LAN TRAFFIC UNDER STABLE PROFILE

Solution	Maximum (Gbps)	Average (Gbps)
PEERSYNC	**5.07**	**0.82**
Kraken [16]	6.61	0.93
Dragonfly [7]	8.80	1.37
Baseline	11.91	6.18

container images, reinforcing PEERSYNC's effectiveness in addressing the challenges of large-scale image distribution.

3) Uplink Occupancy: Tables VII, IX, and X illustrate the cross-network traffic for each image distribution method across three distinct network profiles: stable, congested, and unstable conditions. Cross-network traffic represents data that traverses routers and encounters bandwidth limits, latency, and packet loss, effectively simulating the challenges of data traversing upstream ISP transit networks and the Internet. Unlike local network bandwidth, which is often abundant, cross-network bandwidth is typically constrained by external factors such as physical link limitations and tariff plans. Thus, minimizing cross-network traffic is critical, particularly in edge computing, where the goal is to prioritize intra-LAN resources as much as possible.

- *Sufficient and stable network conditions.* As shown in Table VII, PEERSYNC consistently minimizes cross-network bandwidth usage under stable conditions. By prioritizing local peer data, PEERSYNC achieves an average bandwidth consumption of only 0.82 Gbps, significantly lower than Kraken (0.93 Gbps), Dragonfly (1.37 Gbps), and the Baseline (6.18 Gbps). While Dragonfly and Kraken leverage P2P techniques, their reliance on centralized components for coordination and scheduling results in higher cross-network traffic. The Baseline method incurs the highest bandwidth usage due to its centralized pulling mechanism, which does not take advantage of local data sharing.
- *Congested but stable network conditions.* In congested but stable network conditions (Table IX), PEERSYNC continues to outperform other methods by maintaining an average cross-network traffic of just 0.91 Gbps. The Baseline approach shows the lowest maximum bandwidth (10.36 Gbps), which is a result of its slower and less efficient data retrieval. However, it saturates network capacity with an average bandwidth usage of 9.81 Gbps, underscoring its inefficiency in managing congested networks. Kraken and Dragonfly perform better than the Baseline but still exhibit performance degradation due to their centralized dependencies. Dragonfly, in particular, suffers from its reliance on schedulers and managers, resulting in an average traffic of 4.69 Gbps, over five times higher than PEERSYNC.
- *Congested and unstable network conditions.* Table X highlights the performance of each method under congested and unstable network conditions, characterized by fluctuating packet loss, latency, and bandwidth. PEERSYNC demonstrates remarkable robustness, maintaining the lowest average cross-network traffic at 0.76 Gbps. Kraken performs slightly worse, with an average traffic of 0.89 Gbps, but still manages to leverage its decentralized architecture to reduce reliance on external networks. Dragonfly, on the other hand, suffers significant performance degradation due to its dependence on centralized coordination, resulting in an average traffic of 4.01 Gbps. The Baseline method experiences the lowest maximum bandwidth usage (8.87 Gbps) but continues to exhibit high average traffic (6.34 Gbps), reflecting its inability to adapt to varying network conditions. This highlights the inefficiency of centralized pulling in handling unstable and dynamic environments.

The results across all network profiles clearly demonstrate PEERSYNC's superior ability to minimize cross-network bandwidth consumption. This efficiency is achieved by leveraging local resources and reducing reliance on external network traffic, which is particularly critical in edge environments where bandwidth is constrained and often unstable.

4) Cache Strategy: We evaluated the Cache Cleaner algorithm by analyzing its effectiveness in managing requests and accelerating image retrieval within a LAN. Table VIII shows the relationship between the number of edge devices in a LAN and the average time to download images. Initially, as the number of nodes increases, image distribution times rise slightly due to simultaneous requests to the upstream registry. However, as the LAN becomes more populated, the local cache network is increasingly leveraged, resulting in significantly reduced average retrieval times.

As nodes in the LAN form a collaborative cache, the Cache Cleaner algorithm effectively retains images that are likely to benefit the entire network. This approach optimizes local data sharing and minimizes redundant requests to external networks. The results demonstrate that as the cache network scales, the average download time decreases dramatically, underscoring the efficiency of PEERSYNC's collaborative caching mechanism.

To further validate the Cache Cleaner's performance, we compared its total cache usage against the Least Recently Used (LRU) policy in Table XI. Cache Cleaner consistently achieves better space utilization in multi-node settings by coordinating cache management across neighboring peers. This approach avoids redundant caching of identical content across multiple nodes, enabling more efficient storage utilization within the LAN. In contrast, LRU operates independently and lacks collaboration, leading to higher total cache consumption. Cache Cleaner's ability to prioritize fast intra-LAN retrieval not only optimizes storage but also improves overall efficiency.

5) Distribution of Smaller Images: Although our primary focus is on large AI/ML images, smaller, non-AI base images are also widely used in practical setups. To emulate the operation of Docker Hub, which serves popular images in large volumes daily, we selected the 10 most downloaded images from Docker Hub and increased the request frequency by adjusting the parameter A. Fig. 7 presents the results, demonstrating PEERSYNC's ability to maintain its performance advantage over other P2P-based solutions, even under high-frequency request conditions for smaller images. These results highlight PEERSYNC's capability to handle the significant volumes typical of Docker Hub operations, proving its versatility across diverse workloads.

6) Bursty Requests: In addition to the experiment settings that assume a Poisson arrival process, we evaluate PEERSYNC under a bursty request-arrival scenario, where the request arrival rate temporarily increases. Additionally, we mimic diurnal request patterns by applying bursts of different magnitudes. More specifically, this setting is based on the Congested Network AI/ML image distribution experiment, except that the Poisson rate parameter is varied periodically. The arrival rate increases to $1.5\times$ for 20% of the emulation period, and $3.0\times$ for another 10%, thus periodically applying bursty pressure to the swarm. The evaluation results are given in Fig. 6(d). Empirical results

TABLE VIII
NUMBER OF DEVICES IN A LAN AND THE AVERAGE DISTRIBUTION TIME (OVER 100 RETRIEVAL REQUESTS) SERVED BY LAN CACHE

Number of edge devices within a LAN	1	2	3	4	5	6	7	8	9	10
Average image distribution time (s)	9.11	8.33	9.69	11.34	8.87	8.06	5.08	2.86	2.89	2.19

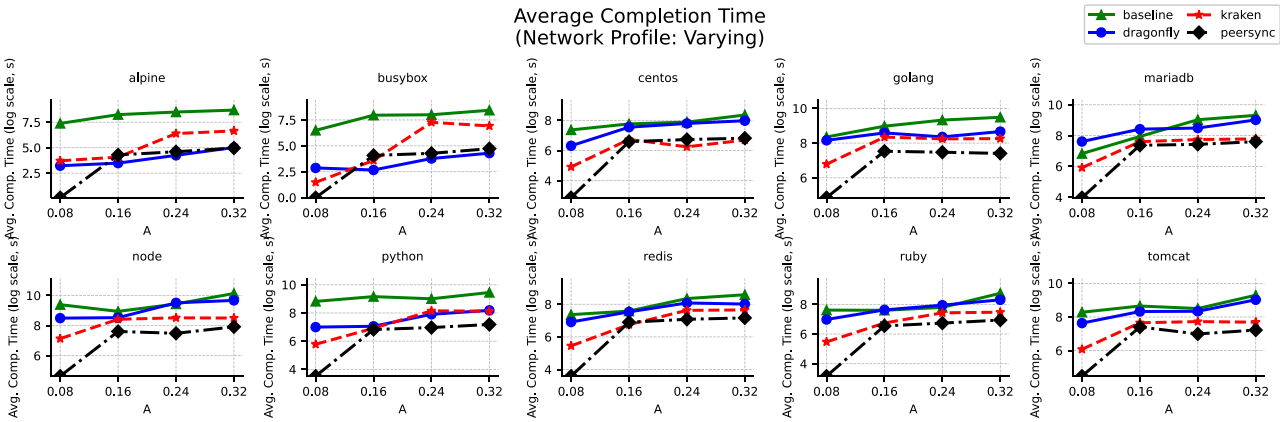

Fig. 7. Average image distribution time under Varying profile (log scale). The x-axis is the parameter A.

TABLE IX
AGGREGATE INTER-LAN TRAFFIC UNDER CONGESTED PROFILE

Solution	Maximum (Gbps)	Average (Gbps)
PEERSYNC	11.50	**0.91**
Kraken [16]	13.76	1.18
Dragonfly [7]	11.94	4.69
Baseline	**10.36**	9.81

TABLE X
AGGREGATE INTER-LAN TRAFFIC UNDER VARYING PROFILE

Solution	Maximum (Gbps)	Average (Gbps)
PEERSYNC	9.15	**0.76**
Kraken [16]	10.99	0.89
Dragonfly [7]	10.13	4.01
Baseline	**8.87**	6.34

show that PEERSYNC is able to maintain consistent high service quality under such bursty and diurnal request arrival patterns.

B. Real-World Experiments

We deployed a physical testbed using Raspberry Pi devices to compare PEERSYNC's performance against the Baseline, Dragonfly, and Kraken. As shown in Fig. 8, the setup comprises six Raspberry Pi (RPi) 4 Model B devices connected via two layer 2 switches, each with a 1 Gbps link speed. Each LAN contains three RPis, with the two switches connected to a common router. The router is configured to forward packets between the networks while limiting the inter-LAN bandwidth to 100 Mbps. To simplify deployment, static routes are configured on the router. This setup is similar to the Docker Compose-based emulation described earlier, with two key differences: (i) Physical hardware and real-world networking scenarios are used; (ii) The aarch64 version of PEERSYNC is deployed instead of the amd64 version to accommodate the ARM-based Raspberry Pi.

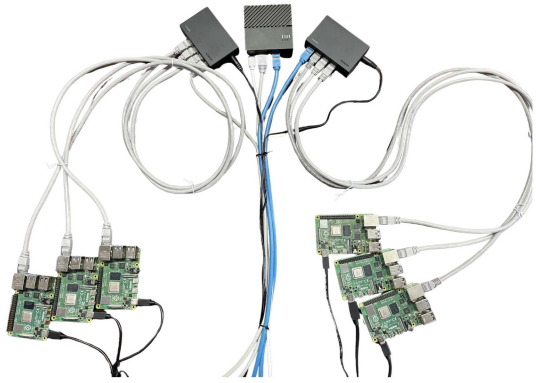

Fig. 8. The testbed for real-world experiments.

Benchmark requests were made to agents running on each RPi to measure image distribution times. Table XII presents the 90th and 99th percentiles of distribution times achieved by each system. The results demonstrate that PEERSYNC significantly outperforms the other systems in real-world conditions.

- At the 90th percentile (P90), PEERSYNC achieves an image distribution time of 190.79 seconds, representing a 39.7% improvement over Kraken, a 60.9% improvement over Dragonfly, and a 75.9% improvement over Baseline.
- At the 99th percentile (P99), PEERSYNC maintains its advantage, with a distribution time of 243.89 seconds, which is 27.8% faster than Kraken, 53.9% faster than Dragonfly, and 69.9% faster than the Baseline.

The results show that PEERSYNC performs better than the other methods significantly in our real-world experiments.

VI. RELATED WORKS

Numerous solutions have been explored to enhance delivery speeds for container images. In this section, we briefly review key developments in this area, covering both container-specific works and methodologies for other resource types.

TABLE XI
NUMBER OF NODES (1 ∼ 10) AND COMPARISON OF THE SUM OF CACHE SPACE OCCUPIED BY CACHE CLEANER AND THE LRU POLICY

Number of edge devices within a LAN	1	2	3	4	5	6	7	8	9	10
Space occupied by Cache Cleaner (MiB)	196	388	475	589	593	668	905	1259	1199	1420
Space occupied by LRU (MiB)	110	206	411	701	837	948	973	1164	1056	1701

TABLE XII
THE 90TH AND 99TH PERCENTILE OF DISTRIBUTION TIMES

Solution	Baseline	Dragonfly	Kraken	PEERSYNC
P90 (s)	790.13	487.50	316.42	**190.79**
P99 (s)	811.23	528.49	337.75	**243.89**

Before the era of edge computing, several pioneering works focused on fast resource delivery in data center networks. VDN [50] leverages hierarchical network topology and shared chunks between virtual machine images to achieve a 30-80× speedup for large virtual machine images under heavy traffic in data center networks. Similarly, VMTorrent [51] improves P2P methods used in live streaming, employing block prioritization, profile-based execution prefetch, on-demand fetch, and decoupling of virtual machine image presentation from the underlying data stream. Experiments show VMTorrent can achieve a 30x speedup over traditional network storage. Since the rise of Docker, containers have become central to deploying workloads in data centers, and distributing container images has emerged as a key issue. While similar to virtual machine images, containers have unique properties, such as a layered structure and registry-based hosting. DevOps practices demand more automated and integrated workload deployment solutions. FID [8], an early P2P-based approach for fast container image distribution, adapts BitTorrent to provide an integrated distribution system. In industry, Dragonfly [7] and Kraken [16] are prominent P2P container image distribution systems. These methods outperform traditional approaches by utilizing spare bandwidth between clients.

However, these solutions do not explicitly address edge computing environments. While Dragonfly [7] and Kraken [16] are P2P-based, they rely on multiple centralized components, which can degrade service quality in the event of component failure. Edge environments are less stable than traditional cloud facilities, and deploying these centralized components in the cloud requires manual configuration across different edge locations. Additionally, edge devices typically have weaker network capabilities, so unrestricted use of P2P protocols can lead to uplink congestion. EdgePier [9], based on IPFS [25], provides a fully decentralized container image distribution system tailored to edge computing. Similarly, Gazzetti et al. [52] proposed a decentralized solution with managers that compute optimal network topologies. For storage-restricted edge devices, learning-based intelligent caching effectively reduces further container spawn latency [53], [54], [55]. While we also employ this incentive in our work, we focus more on providing a full-lifecycle image management solution for systematic optimization of container image management. Recent advancements in edge computing offloading, exemplified by systems like FlexSlice [19], employ RL/TD3 techniques to generate optimal scheduling decisions in fluctuating scenarios. In contrast, PEERSYNC is designed with heuristic approaches that do not involve online learning. This decision is motivated by the significant operational requirements of learning-based models, including access to large training datasets, specialized GPU/NPU, and stable feedback loops that are frequently unavailable in typical edge deployments. Consequently, PEERSYNC's use of a sliding window and RTT-based metrics provides a computationally efficient solution that achieves near-optimal locality, making it better suited for latency-sensitive, one-shot pull operations.

Beyond P2P, another trend involves distributed storage to form a shared storage layer among participating nodes. CoMICon [56], Cider [57], and Wharf [14] use distributed filesystems to accelerate container provisioning. However, these methods are primarily focused on data centers, and implementing distributed storage in edge environments remains challenging [58].

More recently, Starlight [10] has introduced a novel approach by redesigning the container image architecture. It transmits only the necessary components, significantly reducing transfer times. However, Starlight's reliance on specific container engines limits its applicability, and it continues to use a traditional client-server registry architecture, which restricts throughput under high bandwidth loads. Future work could integrate Starlight with PEERSYNC to achieve distributed transmission of large data chunks and selective transmission of necessary data, further improving the containerization ecosystem.

VII. CONCLUDING REMARKS

In this paper, we introduced PEERSYNC, a fully decentralized image distribution system tailored for containerized model inference at the network edge. PEERSYNC leverages P2P downloading to dynamically adapt to changing network conditions and content popularity, significantly outperforming traditional approaches. Its autonomous tracker eliminates single points of failure, enhancing resilience in unstable and resource-constrained edge environments. The integrated cache cleaner also ensures efficient storage use without compromising performance.

Building on the modular approach, future work could also incorporate intelligent caching as on-demand modules. This would allow for dynamic adaptation to various edge site designs, for example, by implementing proactive caching in predictable environments where rich metrics are available. Another promising direction is integrating PEERSYNC with Starlight [10]. Combining PEERSYNC's P2P distribution with Starlight's partial transmission and novel image format could enable a next-generation container ecosystem optimized for edge model inference.

REFERENCES

[1] J. Shah and D. Dubaria, "Building modern clouds: Using Docker, Kubernetes & Google cloud platform," in *Proc. IEEE 9th Annu. Comput. Commun. Workshop Conf.*,2019, pp. 0184–0189.

[2] A. Wang et al., "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC 21)*, Jul. 2021, pp. 443–457. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/wang-ao

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

[4] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *Proc. 3rd USENIX Workshop Hot Top. Edge Comput. (HotEdge 20)*, USENIX Association, Jun. 2020, pp. 126–132. [Online]. Available: https://www.usenix.org/conference/hotedge20/presentation/fu

[5] M. Park, K. Bhardwaj, and A. Gavrilovska, "Toward lighter containers for the edge," in *Proc. 3rd USENIX Workshop Hot Top. Edge Comput. (HotEdge 20)*, Jun. 2020, pp. 148–154. [Online]. Available: https://www.usenix.org/conference/hotedge20/presentation/park

[6] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with KubeEdge," in *Proc. IEEE/ACM Symp. Edge Comput.*, 2018, pp. 373–377.

[7] "Dragonfly," 2017. [Online]. Available: https://d7y.io/

[8] W. Kangjin, Y. Yong, L. Ying, L. Hanmei, and M. Lin, "FID: A faster image distribution system for Docker platform," in *Proc. IEEE 2nd Int. Workshops Found. Appl. Self* Syst. (FAS* W)*, 2017, pp. 191–198.

[9] S. Becker, F. Schmidt, and O. Kao, "Edgepier: P2P-based container image distribution in edge computing environments," in *Proc. IEEE Int. Perform., Comput., Commun. Conf.*,2021, pp. 1–8.

[10] J. L. Chen, D. Liaqat, M. Gabel, and E. de Lara, "Starlight: Fast container provisioning on the edge and over the WAN," in *Proc. 19th USENIX Symp. Networked Syst. Des. Implementation (NSDI 22)*, Renton, WA, USA: USENIX Association, Apr. 2022, pp. 35–50. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/chen-jun-lin

[11] N. Zhao et al., "Slimmer: Weight loss secrets for Docker registries," in *Proc. IEEE 12th Int. Conf. Cloud Comput.*, 2019, pp. 517–519.

[12] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy Docker containers," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST 16)*, Santa Clara, CA, USA: USENIX Association, Feb. 2016, pp. 181–195. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter

[13] H. Li, Y. Yuan, R. Du, K. Ma, L. Liu, and W. Hsu, "DADI: Block-level image service for agile and elastic application deployment," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC 20), USENIX Assoc.*, Jul. 2020, pp. 727–740. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/li-huiba

[14] C. Zheng et al., "Wharf: Sharing Docker images in a distributed file system," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 174–185.

[15] M. Liang, S. Shen, D. Li, H. Mi, and F. Liu, "HDID: An efficient hybrid Docker image distribution system for datacenters," in *Proc. Softw. Eng. Methodol. Emerg. Domains: 15th Nat. Softw. Application Conf.*, NASAC 2016, Kunming, Yunnan, Springer, 2016, pp. 179–194.

[16] Uber, "Kraken," 2018. [Online]. Available: https://github.com/uber/kraken

[17] A. L. Jia and D. M. Chiu, "Designs and evaluation of a tracker in P2P networks," in *Proc. Eighth Int. Conf. Peer-to-Peer Comput.*, 2008, pp. 227–230.

[18] J. Tate et al., *Introduction to Storage Area Networks*. Poughkeepsie, NY: IBM Redbooks, 2018.

[19] A. Mohajer, J. Hajipour, and V. C. Leung, "Dynamic offloading in mobile edge computing with traffic-aware network slicing and adaptive TD3 strategy," *IEEE Commun. Lett.*, vol. 29, no. 1, pp. 95–99, Jan. 2024.

[20] Y. Chen, S. Deng, H. Zhao, Q. He, Y. Li, and H. Gao, "Data-intensive application deployment at edge: A deep reinforcement learning approach," in *Proc. IEEE Int. Conf. Web Serv.,*2019, pp. 355–359.

[21] H. Zhao, S. Deng, C. Zhang, W. Du, Q. He, and J. Yin, "A mobility-aware cross-edge computation offloading framework for partitionable applications," in *Proc. IEEE Int. Conf. Web Serv.*, 2019, pp. 193–200.

[22] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar, "Distributed redundant placement for microservice-based applications at the edge," *IEEE Trans. Serv. Comput.*, vol. 15, no. 3, pp. 1732–1745, May/Jun. 2022.

[23] B. Carlsson and R. Gustavsson, "The rise and fall of Napster-an evolutionary approach," in *Proc. Int. Comput. Sci. Conf. Act. Media Technol.*, 2001, pp. 347–354.

[24] "Bittorrent," 2001. [Online]. Available: https://www.bittorrent.org

[25] J. Benet, "Ipfs-content addressed, versioned, p2p file system," 2014, *arXiv:1407.3561*.

[26] B. Cohen, "The BitTorrent protocol specification," Jan. 2008. [Online]. Available: https://www.bittorrent.org/beps/bep_0003.html

[27] G. Neglia, G. Reina, H. Zhang, D. Towsley, A. Venkataramani, and J. Danaher, "Availability in BitTorrent systems," in *Proc. IEEE INFOCOM 2007-26th IEEE Int. Conf. Comput. Commun.*, 2007, pp. 2216–2224.

[28] A. Kononova, A. Gorodilov, A. K. Myo, and L. Gagarina, "Lifecycle and survival rate of torrent trackers," in *Proc. IEEE Conf. Russian Young Researchers Elect. Electron. Eng.,*2021, pp. 2133–2136.

[29] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: Automatically debloating containers," in *Proc. 11th Joint Meeting Foundations Softw. Eng.*, 2017, pp. 476–486.

[30] "Slimtoolkit," 2015. [Online]. Available: https://slimtoolkit.org/

[31] "Huggin face safetensors," 2024. [Online]. Available: https://huggingface.co/safetensors/

[32] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, vol. 32, pp. 8026–8037.

[33] A. Dubey et al., "The llama 3 herd of models," 2024, *arXiv:2407.21783*.

[34] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *Commun. ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[35] M. A. Kafi, D. Djenouri, J. Ben-Othman, and N. Badache, "Congestion control protocols in wireless sensor networks: A survey," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 3, pp. 1369–1390, 3rd Quart. 2014.

[36] S. Agarwal, A. Krishnamurthy, and R. Agarwal, "Host congestion control," in *Proc. ACM SIGCOMM 2023 Conf.*, 2023, pp. 275–287.

[37] S. Arslan, Y. Li, G. Kumar, and N. Dukkipati, "Bolt:{Sub-RTT} congestion control for {Ultra-Low} latency," in *Proc. 20th USENIX Symp. Networked Syst. Des. Implementation* (NSDI 23), 2023, pp. 219–236.

[38] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[39] T. Panitanarak, "Scalable single-source shortest path algorithms on distributed memory systems," in *Soft Computing in Data Science*, B. W. Yap, A. H. Mohamed, and M. W. Berry, Eds. Singapore: Springer Singapore, 2019, pp. 19–33.

[40] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Conf. Operating Syst. Des. Implementation (OSDI'06)*, 2006, pp. 307–320.

[41] A. Makris, I. Kontopoulos, E. Psomakelis, S. N. Xyalis, T. Theodoropoulos, and K. Tserpes, "Performance analysis of storage systems in edge computing infrastructures," *Appl. Sci.*, vol. 12, no. 17, 2022, Art. no. 8923.

[42] A. J. Smith, "Cache memories," *ACM Comput. Surv.,* vol. 14, no. 3, pp. 473–530, 1982.

[43] K. So and R. N. Rechtschaffen, "Cache operations by MRU change," *IEEE Trans. Comput.*, vol. 37, no. 6, pp. 700–709, Jun. 1988.

[44] I. Granite Team, "Granite 3.0 Language Models," 2024. [Online]. Available: https://github.com/ibm-granite/granite-3.0-language-models

[45] A. Kirillov et al., "Segment anything," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2023, pp. 4015–4026.

[46] "Langchain llm app development framework," 2022. [Online]. Available: https://langchain.com/

[47] M. Abadi et al., "{TensorFlow}: A system for {Large-Scale} machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation* (OSDI 16), 2016, pp. 265–283.

[48] "tc(8)-linux man page," 2001. [Online]. Available: https://linux.die.net/man/8/tc

[49] "iPerf - the ultimate speed test tool for TCP, UDP and SCTP," 2014. [Online]. Available: https://iperf.fr/

[50] C. Peng, M. Kim, Z. Zhang, and H. Lei, "VDN: Virtual machine image distribution network for cloud data centers," in *Proc. Proc. IEEE INFOCOM*, 2012, pp. 181–189.

[51] J. Reich et al., "VMTorrent: Scalable P2P virtual machine streaming," *CoNEXT*, vol. 12, pp. 289–300, 2012.

[52] M. Gazzetti, A. Reale, K. Katrinis, and A. Corradi, "Scalable linux container provisioning in fog and edge computing platforms," in *Proc. Euro-Par 2017: Parallel Process. Workshops: Euro-Par 2017 Int. Workshops*, Santiago de Compostela, Spain, Aug. 28-29, 2017, Revised Selected Papers 23. Springer, 2018, pp. 304–315.

[53] A. Chen and G. Ishigaki, "Scaling container caching to larger networks with multi-agent reinforcement learning," in *Proc. 33 rd Int. Conf. Comput. Commun. Netw.*2024, pp. 1–5.

[54] D. Jayaram, S. Jeelani, and G. Ishigaki, "Container caching optimization based on explainable deep reinforcement learning," in *Proc. GLOBECOM 2023-2023 IEEE Glob. Commun. Conf.*, 2023, pp. 7127–7132.

[55] H. Torabi, H. Khazaei, and M. Litoiu, "A learning-based caching mechanism for edge content delivery," in *Proc. 15th ACM/SPEC Int. Conf. Perform. Eng.*, 2024, pp. 236–246.

[56] S. Nathan, R. Ghosh, T. Mukherjee, and K. Narayanan, "Comicon: A co-operative management system for Docker container images," in *Proc. IEEE Int. Conf. Cloud Eng.,*2017, pp. 116–126.

[57] L. Du, T. Wo, R. Yang, and C. Hu, "Cider: A rapid Docker container deployment system through sharing network storage," in *Proc. IEEE 19th Int. Conf. High Perform. Comput. Commun.; IEEE 15th Int. Conf. Smart City; IEEE 3 rd Int. Conf. Data Sci. Syst.* (HPCC/SmartCity/DSS), 2017, pp. 332–339.

[58] A. Makris, E. Psomakelis, T. Theodoropoulos, and K. Tserpes, "Towards a distributed storage framework for edge computing infrastructures," in *Proc. 2nd Workshop Flexible Resource Appl. Manage. Edge*, 2022, pp. 9–14.

Yinuo Deng received the MS degree from the College of Computer Science and Technology, Zhejiang University, Hangzhou, China, in 2025 and the BS degree from the School of Artificial Intelligence, Beijing University of Posts and Telecommunications, Beijing, China, in 2022. He is currently with Alibaba Cloud, Hangzhou, China. His research interests include cloud computing, networking, and distributed systems.

Hailiang Zhao (Member, IEEE) received the PhD degree in computer science from Zhejiang University, in 2024, with a visiting research appointment with Nanyang Technological University, Singapore from 2022 to 2023. He is currently a ZJU 100 young professor with the School of Software Technology, Zhejiang University, and an Outstanding Qizhen Young Scholar. He has authored or coauthored over 40 papers in leading journals and conferences, including *Proceedings of the IEEE*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Services Computing*, NeurIPS, CVPR, and ICWS. His research interests include the intersection of services computing and service system performance optimization, with a focus on developing intelligent, learning-augmented algorithms and systems. He was a regular reviewer for prestigious venues such as *IEEE Transactions on Services Computing*, *ACM Transactions on Knowledge Discovery from Data*, Chinese Journal of Computers, FGCS, NeurIPS, and CVPR. He was the recipient of the several honors, including Incentive Program for Outstanding Ph.D. Dissertations by CCF-TCSC (2025), Zhejiang University Outstanding Doctoral Dissertation Award (2024), and Best Student Paper Award at IEEE ICWS 2019.

Dongjing Wang (Member, IEEE) received the BS and PhD degrees in computer science from Zhejiang University, Hangzhou, China, in 2012 and 2018, respectively. He was cotrained with the University of Technology Sydney, Australia, for one year. He is currently an associate professor with Hangzhou Dianzi University, Hangzhou. He has authored or coauthored more than 70 journal articles, including *IEEE Transactions on Multimedia*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Services Computing*, *IEEE Transactions on Cybernetics*, *IEEE Transactions on Neural Networks and Learning Systems*, *ACM Transactions on Information Systems*, and refereed conferences. His research interests include recommender systems, machine learning, data mining, and business process management.

Peng Chen received the BS degree in computer science from Soochow University, China, and the MS degree in informatics from Kyoto University, Japan, in 2020. He is currently working toward the PhD degree with the College of Computer Science and Technology, Zhejiang University, China. He has authored or coauthored papers in venues such as NeurIPS, *IEEE Transactions on Services Computing*, and *IEEE/ACM Transactions on Networking*. His research interests include theoretical machine learning, distributed systems, and service computing.

Wenzhuo Qian received the BS degree from the School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China, in 2023. He is currently working toward the Phd's degree with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. His research interests include edge computing and service computing.

Jianwei Yin received the PhD degree in computer science from Zhejiang University (ZJU), in 2001. He was a visiting scholar with the Georgia Institute of Technology. He is currently a full professor with the College of Computer Science, ZJU. He has authored or coauthored more than 100 papers in top international journals and conferences. His current research interests include service computing and business process management. He is also an associate editor of *IEEE Transactions on Services Computing*.

Schahram Dustdar (Fellow, IEEE) is currently a full professor of computer science (informatics) with a focus on Internet Technologies heading the Distributed Systems Group with the TU Wien. He was founding co-Editor-in-Chief of *ACM Transactions on Internet of Things* (ACM TIoT). He is also the Editor-in-Chief of Computing (Springer). He is alo an associate editor of *IEEE Transactions on Services Computing*, *IEEE Transactions on Cloud Computing*, *ACM Computing Surveys*, *ACM Transactions on the Web*, and *ACM Transactions on Internet Technology*, as well as on the editorial board of *IEEE Internet Computing* and IEEE Computer. He is the recipient of multiple awards: TCI Distinguished Service Award (2021), IEEE TCSVC Outstanding Leadership Award (2018), IEEE TCSC Award for Excellence in Scalable Computing (2019), ACM Distinguished Scientist (2009), ACM Distinguished Speaker (2021), IBM Faculty Award (2012). He is an elected member of the Academia Europaea: The Academy of Europe, where the chairperson of the Informatics Section for multiple years. He is an IEEE fellow (2016), an Asia-Pacific Artificial Intelligence Association (AAIA) President (2021) and fellow (2021). He is an EAI fellow (2021) and an I2CICC fellow (2021). He is a Member of the IEEE Computer Society Fellow Evaluating Committee (2022 and 2023).

Shuiguang Deng (Senior Member, IEEE) received the BS and PhD degrees in computer science, with the College of Computer Science and Technology, Zhejiang University, China, in 2002 and 2007, respectively, where he is currently a full professor. He was with the Massachusetts Institute of Technology, in 2014 and Stanford University, in 2015 as a visiting scholar. He has authored or coauthored more than 100 papers in journals and refereed conferences. His research interests include edge computing, service computing, cloud computing, and business process management. He was for the journal *IEEE Transactions on Services Computing*, *Knowledge and Information Systems, Computing*, and *IET Cyber-Physical Systems: Theory & Applications* as an associate editor. In 2018, he was granted the Rising Star Award by IEEE TCSVC. He is a fellow of IET and a senior member of IEEE.