CacheFS: Peer-to-Peer Cooperative Caching for LLM Weight Distribution at Cloud Scale

Zheng Liu
Zhejiang University
Hangzhou, China
Alibaba Group
Hangzhou, China
wenqing.lz@alibaba-inc.com

Eryu Guan
Alibaba Group
Hangzhou, China
eryu.gey@alibaba-inc.com

Shang Zhao
Alibaba Group
Hangzhou, China
zhaoshang.zs@alibaba-inc.com

Chengkun Wei✉
Zhejiang University
Hangzhou, China
weichengkun@zju.edu.cn

Teng Ma
Alibaba Group
Hangzhou, China
sima.mt@alibaba-inc.com

Jin Shi
Zhejiang University
Hangzhou, China
shijin4300@zju.edu.cn

Ye Cao
Alibaba Group
Hangzhou, China
caoye.cao@alibaba-inc.com

Chen Cao
Alibaba Group
Hangzhou, China
caochen.cao@alibaba-inc.com

Wenpeng Li
Alibaba Group
Hangzhou, China
xingke.lwp@alibaba-inc.com

Tao He
Alibaba Group
Hangzhou, China
linzhu.ht@alibaba-inc.com

Wenyuan Yu
Alibaba Group
Hangzhou, China
wenyuan.ywy@alibaba-inc.com

Tao Ma✉
Alibaba Group
Hangzhou, China
boyu.mt@alibaba-inc.com

Wenzhi Chen
Zhejiang University
Hangzhou, China
chenwz@zju.edu.cn

Abstract

Elastic LLM inference must bring new GPU instances online quickly, but burst scale-out makes many instances fetch the same large model from shared cloud storage. In our production environment, distributing Qwen2.5-72B from NFS to 50 instances takes 28 minutes. Model weights are immutable, read in full, loaded nearly sequentially, and identical across replicas. These properties allow inference nodes to use host memory and intra-cluster bandwidth as a cooperative cache without the write and consistency mechanisms of a general-purpose file system.

We present **CacheFS**, a peer-to-peer cooperative caching file system. Its central mechanism is hash-aware cooperative eviction. Under memory pressure, each node preferentially retains blocks assigned to it by consistent hashing. When aggregate cache capacity covers the working set, the union of N partial caches converges toward full model coverage even if no node can hold the complete model. This requires neither centralized nor per-block retention coordination. CacheFS also provides a coroutine-based data plane, proactive preload, a pre-warmed CacheRoot tier, and a transparent FUSE interface for unmodified inference frameworks. After more than eighteen months of production operation across four cloud regions, CacheFS distributes Qwen2.5-72B to 50 instances in 30 seconds, **56**× faster than NFS. For DeepSeek-R1, per-node loading from object storage decreases from 19 minutes to 200 seconds. Together with a companion fast-startup runtime, CacheFS saves more than **7,700** GPU card-hours per day.

CCS Concepts: • **Computer systems organization** → **Peer-to-peer architectures**; *Cloud computing*; • **Software and its engineering** → **File systems management**.

Keywords: peer-to-peer caching, distributed file systems, LLM inference, model distribution, elastic scaling, RDMA

ACM Reference Format:
Zheng Liu, Eryu Guan, Shang Zhao, Chengkun Wei, Teng Ma, Jin Shi, Ye Cao, Chen Cao, Wenpeng Li, Tao He, Wenyuan Yu, Tao Ma, and Wenzhi Chen. 2026. CacheFS: Peer-to-Peer Cooperative Caching for LLM Weight Distribution at Cloud Scale. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15-18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3828523.3857016



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857016

1 Introduction

Elastic scaling is the defining operational challenge of cloud LLM inference because request volumes fluctuate by an order of magnitude within minutes, forcing platforms to bring tens of GPU-equipped instances online rapidly. The dominant cold-start bottleneck is *model weight retrieval*. A 70B-parameter model occupies 140 GB in FP16, and current frontier models at the 200B–700B scale reach 400–650 GB. Under the default storage configurations in our production environment, retrieving a 70B model can take about nine minutes on one node (§2.2). Burst scale-out compounds the problem because N instances competing on shared upstream bandwidth degrade to $O(N)$ startup time, forcing operators to keep large pools of idle GPU instances on standby just to absorb scale-out bursts, defeating the cost rationale for elastic provisioning in the first place. Inference servers also provide resources for caching. Host memory typically exceeds GPU capacity by 2–8×, although KV-cache offloading increasingly competes for it, and intra-cluster networks provide more bandwidth than upstream storage (§2.4).

Existing systems address fragments of this with disproportionate complexity. Distributed file systems (Ceph [39], JuiceFS [23]) provide read-write POSIX through centralized metadata services. In our deployment, a model load opens tens of large weight files and reads them near-sequentially, so centralized metadata is not a throughput bottleneck. The difference is the operational complexity and read-write machinery that model loading does not require. Dragonfly [8] accelerates P2P container-image distribution but exposes no file-system interface and requires a heavyweight coordinator stack.

Key insight. We observe that LLM model loading has a constrained I/O profile in which files are *read-only*, consumed *in their entirety*, *near-sequentially*, and *identically* across replicas. Specializing for these properties exposes one design question: when each node has only enough memory to cache part of the model, how should the partial caches be coordinated so the cluster behaves like a single, model-sized cache? We present **CacheFS**[1], a peer-to-peer cooperative caching file system whose core mechanism is **hash-aware cooperative eviction**. These properties allow CacheFS to omit write consistency, partial-read optimizations, and support for heterogeneous access patterns. Under memory pressure, each node deterministically retains the blocks for which it is the consistent-hash owner. Without centralized or per-block retention coordination, the union of partial caches converges toward full model coverage. The cluster can therefore serve every block from a peer even when no node can hold the complete model. In production, CacheFS uses this cluster-level coverage for peer-served scale-out and achieves a 56× speedup over the configured NFS path (§5.2). Section 5.8 shows this eliminates upstream backsource entirely on a workload where a uniformly-random eviction policy (2-random) re-fetches 800+ blocks per node. Several supporting mechanisms make this insight usable, including a sourceless distribution model that lets consumers without storage access join the cooperative cache, gossip coordination that requires no external etcd, ZooKeeper, or Redis service in the user cluster, a warm CacheRoot tier that breaks the bootstrap paradox, and a coroutine-based data plane that drives the cluster's commodity TCP NICs to line rate (with optional RDMA for low-latency deployments).

Contributions. (1) *An eighteen-month production deployment* of CacheFS on a major cloud inference platform spanning a fleet at the hundred-thousand-GPU scale and four regions, including 56× faster Qwen2.5-72B distribution to 50 instances than NFS and an 82.5% reduction in DeepSeek-R1 per-node loading relative to object storage, with 7,700+ GPU card-hours saved per day in conjunction with a companion fast-startup runtime that reduces framework initialization (§5). (2) *A production-driven design* combining sourceless P2P distribution, gossip-based coordination, proactive preload, a coroutine-based TCP/RDMA data plane, and a CacheRoot warm tier behind a transparent FUSE interface (§3, §4). (3) *Hash-aware cooperative eviction*, the central mechanism that makes a cluster's partial caches converge toward full model coverage without coordination. Under pressure this drives upstream backsource to zero where the 2-random eviction policy still re-fetches 800+ blocks per node (§3.4, §5.8). (4) *Operational lessons*, including non-obvious bottlenecks (Go GC, gossip fan-out) and deployment-time interventions (preload, priming-wave practice) that we report so other operators can avoid the same pitfalls (§6).

2 Background and Motivation

This section describes the deployment context of LLM inference services, characterizes the cold-start bottleneck that motivates CacheFS, and identifies the workload properties and cluster resources that inform its design.

2.1 LLM Inference in the Cloud

Modern inference platforms run on Kubernetes [5], with weight files stored on cloud-managed networked storage (NFS or object storage). On one major cloud, the inference GPU fleet grew roughly five-fold over fourteen months into a multi-region deployment at the hundred-thousand-GPU scale. The production service provisions 1–5 Gbps of storage bandwidth per client based on cost and hardware constraints, far below GPU-node memory bandwidth. This gap becomes important when hundreds of GB must be delivered quickly.

Real elastic workload. Figure 1 plots a 24-h trace of one production tenant. Instance count swings 29–155 (5.3×), and

[1] Our deployed system is unrelated to the earlier CacheFS name used for local caching of network-file-system data.

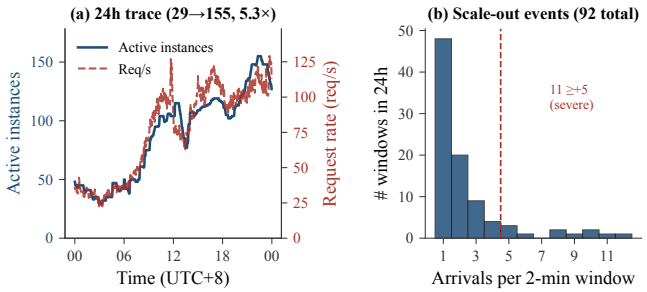


Figure 1. Production trace from one inference tenant (24-h window, 2-min samples). (a) Active instance count swings 5.3× diurnally, and request rate tracks it. (b) The day contains 92 scale-out events. Eleven (12%) admit ≥ 5 instances within a 2-min window. At this burst size, N-way contention on upstream NFS dominates startup latency.

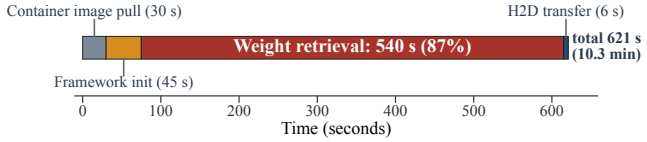


Figure 2. Cold-start pipeline breakdown for a 70B model loaded from cloud NFS. Weight retrieval dominates at 87% of total startup time. Framework initialization (30–60s) is the second largest phase.

request rate swings 22.5–129 req/s (5.7×). Of the 720 two-minute windows, 92 (12.8%) contain at least one scale-out event. Among them, 24 admit ≥ 3 new instances, 11 admit ≥ 5, and the largest single window adds +12. This tenant alone schedules **220 instance-arrivals/day**. At the 2 Gbps NFS cap, each Qwen2.5-72B (136 GB) load takes 540s. When a pod lands on a node without a warm local cache, its eight GPUs remain idle during loading. This wastes up to $220 \times 540s \times 8 \approx 264$ GPU-card-hours/day on this tenant alone. A sustained morning ramp 31⟶104 instances (3.4× in 9 h) shows weight loading dominates every ramp, not just peaks.

2.2 Anatomy of Cold-Start Latency

A cold start runs four phases sequentially: (1) container image pull (tens of seconds), (2) framework initialization, including CUDA warmup and shard planning (30–60s before the first weight byte, as Figure 2 shows), (3) **weight retrieval** (approximately nine minutes per node for a 70B/140 GB model over 2 Gbps NFS, with a 235B/470 GB model scaling proportionally), and (4) host-to-device transfer (less than 6s for 140 GB over PCIe Gen4 at 25 GB/s). Phase 3 dominates at ~87% of total time (Figure 2).

The problem worsens during *burst scale-out*. When N pods simultaneously request the same model, they share the upstream bandwidth of the storage backend. If the NFS service provisions a total of B_{total} Gbps across all clients, each pod effectively receives B_{total}/N, causing startup time to scale linearly with the burst size. Production measurements confirm this severity. Distributing Qwen2.5-72B to 50 simultaneous instances from cloud NFS requires approximately 28 minutes end-to-end, while loading a DeepSeek-R1 model takes about 19 minutes per node. Operators compensate by maintaining warm standby pools of idle GPU instances, a costly strategy that wastes scarce accelerator resources.

2.3 LLM File Access Characteristics

LLM inference I/O has four properties unusual among conventional file-system workloads. *Read-only:* weight files are immutable post-publication, eliminating write coordination, consistency protocols, and journaling. *Full-file:* each model file is read in its entirety, not as a working set of hot regions. A serving replica must load every weight file before it can serve requests. Lazy delivery of an initial working set does not remove this requirement. *Near-sequential:* `safetensors` [22] memory-maps the header and sequentially loads tensors per the metadata index, with occasional small seeks at tensor boundaries, highly amenable to aggressive prefetching. *Homogeneous, repeated:* every replica reads the same files in the same order, so a block fetched by any node can serve all subsequent requesters. For the tensor and model formats in our deployment, successive versions have little byte-identical content in common. A new mount reads fresh metadata for an updated model. Changes in path, length, or `mtime` produce new cache keys, so old blocks are not reused (§3.7). The useful reuse is therefore among replicas in the same burst. Hash-aware eviction lets their aggregate cache cover the complete model (§3.4). Together, these properties allow a tailored system to dispense with write paths, locking, and metadata consistency.

2.4 Resource Opportunities in Inference Clusters

Inference clusters provide two resources for faster model distribution. GPU servers typically have 2–8× more host DRAM than GPU memory (1–2 TB alongside 640 GB), while intra-cluster networks reach up to 400 Gbps. Host DRAM is also used by the inference runtime, and long-context serving increasingly uses it for KV-cache offloading. The remaining capacity can still buffer immutable model weights and reduce repeated reads from upstream storage.

2.5 Limitations of Existing Solutions

LLM weight loading requires a different combination of mechanisms from existing content-distribution systems. Distributed file systems such as Ceph [39] and JuiceFS [23] provide read-write semantics through separate metadata services. In our deployment, a model load opens tens of large weight files and reads them near-sequentially, so centralized metadata is not a throughput bottleneck. The difference is the additional operational complexity and read-write support, which model loading does not need. Dragonfly [8]

delivers P2P at the HTTP layer with a centralized scheduler, exposes no POSIX, and targets layered OCI images. Local SSD caching atop NFS shares nothing across nodes and faces a *bootstrap paradox* because the first wave has empty caches exactly when fast startup matters most.

3 System Design

This section describes CacheFS's architecture, sourceless distribution, cluster management, cooperative caching, data transfer, CacheRoot, and correctness mechanisms.

3.1 Overview

CacheFS aggregates idle host memory across inference cluster nodes into a cooperative P2P cache for model weight files. Figure 3 illustrates the overall architecture.

Each CacheFS node is structured as a layered stack. Three access interfaces serve different deployment scenarios: a *FUSE mount* provides POSIX compatibility for unmodified inference frameworks (vLLM, SGLang, Transformers); a *native SDK* (C/Python) offers direct block-level access for latency-sensitive integrations; and an optional *daemon* serves as a shared caching backend for co-located containers (§4.1). CacheFS can cache any POSIX-accessible directory regardless of the backing store (cloud NFS, object storage via a FUSE gateway, or a local file system), requiring no changes to the upstream storage configuration. Upstream durable storage remains the system of record for training and model updates. It also provides the final source on a cache miss. Below the VFS, the metadata engine and cache store implement the five-layer lookup (local → P2P peers → seeders → CacheRoot → source). The transport layer separates control (Serf + gRPC [18]) from data (coro-rpc over TCP/RDMA).

End-to-end walkthrough. In a 50-instance Qwen2.5-72B scale-out, all 50 pods FUSE-mount CacheFS at t=0, lazily exposing the upstream NFS namespace. vLLM then reads `model-XXXX-of-37.safetensors` via FUSE. On each miss, the cache store walks the lookup chain. A per-node pending-operation table coalesces concurrent requests (§4.3), but it does not coordinate misses across nodes. Before a new ownership announcement reaches other nodes, several pods may fetch the same missing block from upstream. Each completed fetch caches the block and announces ownership, so later reads can use peers. Gossip shortens this duplicate-fetch window but does not eliminate it. Read-ahead and (when scheduled) preload warm the P2P cache faster than the application consumes it. Within a few gossip rounds every block is held by multiple peers, so most reads resolve at intra-cluster bandwidth without touching upstream, completing the 50-instance burst in 30s on 32 Gbps cloud NICs (§5.2). Because epidemic propagation allows each node to fetch from multiple peers simultaneously, batch completion time approximates the single-node load time rather than scaling with N.

3.2 Sourceless Distribution Model

CacheFS additionally supports a *sourceless* mode for deployments where consumers have no upstream storage access (e.g., a fine-tune producer that must deliver weights to consumers without round-tripping through cloud storage). A **provider** walks a local directory at startup, caches its data, and broadcasts a Serf readiness event. **Consumers** join the cluster, receive the inode tree via a single gRPC `LoadMeta` call, and thereafter serve both metadata and data to later joiners, spreading epidemically with effective bandwidth doubling per round (yielding $O(\log N)$ warm-up, §5.4). The invariant is $N \cdot C \geq B$, where N is the number of nodes, C is the per-node cache capacity, and B is the total working-set size. Under this invariant, the union of partial caches must cover the working set. Because gossip propagation is eventually consistent, sourceless mode offers best-effort availability, and transient membership churn or extreme cache pressure can make blocks unavailable. If a requested block cannot be retrieved from the cooperative cache, the read returns an error rather than falling back to upstream storage.

Sourceless mode was motivated by an early fast container-startup scenario. In current production, the source-backed path (§3.6) handles all model distribution because it provides stronger availability guarantees when the cluster contracts or experiences correlated failures.

3.3 Gossip-Based Cluster Management

CacheFS uses Serf [21] (a SWIM [12] implementation) for decentralized membership and application-level events. Nodes discover each other through a join address or K8s service discovery. The SWIM failure detector identifies crashed or partitioned nodes within seconds. *Group isolation* via distinct Serf-group tags lets multiple independent CacheFS instances coexist on the same network without cross-talk.

To track block ownership, each node broadcasts an ANNOUNCE on cache insertion and a REVOKE on eviction. Each peer's local `metaMap` records, per `(target, block)`, the set of hosts that hold it.

gRPC Tracker Exchange (§5.9). New joiners obtain the membership list through Serf, then directly exchange tracker information with each existing peer via a lightweight gRPC call, avoiding the broadcast storm that an all-members re-announce would cause at scale. This replaces the earlier event-based tracker bootstrap (§6.1).

3.4 Cache Hierarchy and Hash-Aware Eviction

Multi-layer hierarchy. CacheFS resolves reads in strict layer order: (1) *local cache* (memory-backed via `memfd` for zero-copy or disk-backed; default 128 GiB, 4–128 MiB blocks); (2) *P2P peers* consulted via the metaMap (up to 10 candidates, failed peers blacklisted briefly); (3) *seeders* for full-copy holders when the metaMap lacks the block; (4) *CacheRoot* (§3.6); (5) *upstream source* as last resort. Each fetched block is cached

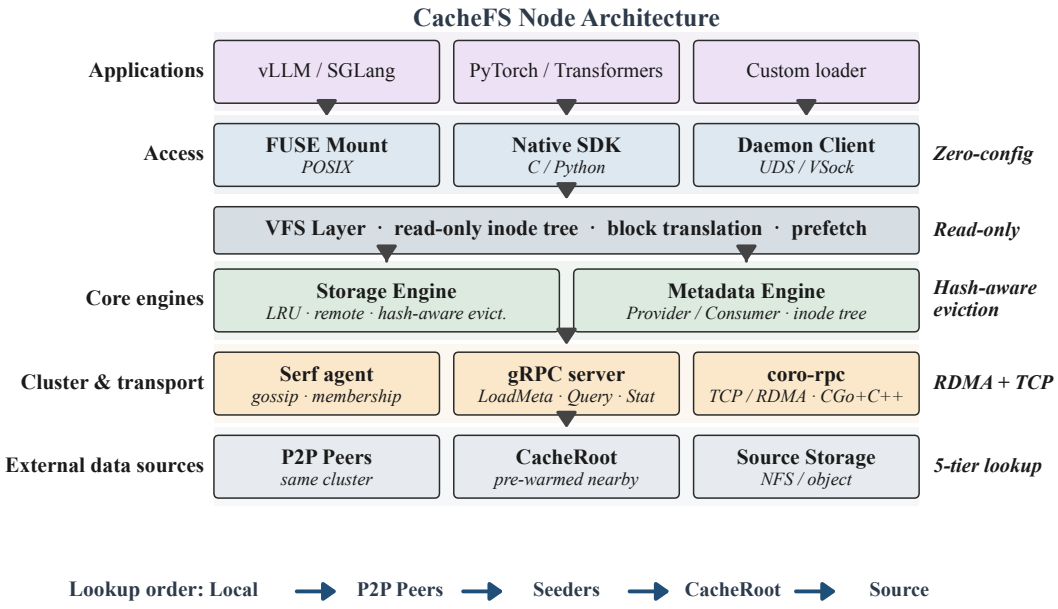


Figure 3. CacheFS architecture overview. Each node exposes three access interfaces (FUSE, SDK, Daemon) atop a read-only VFS. The cache store and metadata engine are served by three transport channels: Serf gossip (control plane), gRPC (metadata), and coro-rpc (data plane). Cache lookups proceed left-to-right through five layers: local cache, P2P peers, seeder nodes, CacheRoot, and the remote source.

locally and gossiped, so subsequent requests resolve at the P2P layer.

Prefetch and preload. CacheFS prefetches 24 blocks ahead of the read position by default, scaled to available buffer memory. Read-driven prefetch only fires once the application starts reading, but inference frameworks spend 30–60s on container start and CUDA init before the first weight byte. CacheFS therefore adds a `preload` operation that fetches a target into the local cache (and, with `--broadcast`, into every cluster node) as soon as the mount is ready. Preload reuses the same multi-layer lookup. A 60s preload window reduces per-instance read time by 7–41% (mean 27%) across our 8-wave scale-out (§5.6).

Hash-aware eviction. A cooperative cache should maximize the *aggregate* distinct blocks across the cluster. CacheFS uses a consistent hash ring (100 virtual nodes per physical node, refreshed on membership change) to assign each block a *responsible node* $R(\cdot)$. At node i, eviction runs five progressive rounds. When hash-aware mode is enabled and the cluster has more than one node, rounds 1–3 progressively delete blocks for which i is not responsible, first using pair-compare and then direct eviction. If these rounds cannot free enough space, rounds 4–5 fall back to 2-random eviction over all blocks.

Convergence. Consistent hashing spreads blocks roughly evenly, making each node responsible for approximately B/N blocks (where B is the total block count and N is the number of nodes). Because rounds 1–3 preferentially evict non-responsible blocks, each node retains at least its responsible share. Provided the per-node cache capacity $C \geq B/N$, the cluster's combined cache covers the entire working set even when no single node can hold the model ($NC \approx B$). §5.8 confirms this at $NC \approx B$ (60 GiB cache, 136 GB model, N=3→4). On the warming wave, hash-aware eviction drives backsource fetches to zero, while 2-random re-fetches 800+ blocks per node.

3.5 Data Transfer Plane

CacheFS separates its control plane (Serf + gRPC for metadata) from its data plane, which must saturate the NIC. The data plane uses `coro-rpc`[2], a coroutine-based RPC layer that bridges Go to a C++20 coroutine library via CGo and multiplexes block transfers per connection. Over TCP, coro-rpc sustains 140 Gbps on a single 200 Gbps NIC as a multi-connection aggregate. Production-class cloud Ethernet runs at 10–48 Gbps per host depending on instance class, so the data path is rarely the limiter. An optional RDMA (RoCE v2) path sustains 360 Gbps on a single 400 Gbps NIC and 485 Gbps with two such NICs aggregated. Beyond two NICs, aggregate throughput plateaus. Initial profiling suggests memory pages not being pre-faulted as the likely bottleneck, and we leave further scaling as open work. We use the RDMA path for latency-critical fast container-startup scenarios on RDMA-provisioned nodes. Transport selection is per-connection

[2] https://github.com/alibaba/yalantinglibs/tree/main/src/coro_rpc

with transparent TCP fallback on RDMA init failure. CacheFS enforces per-connection bandwidth caps to prevent data transfers from starving co-located inference traffic. All transferred blocks are CRC32-verified before cache insertion.

3.6 CacheRoot: Near-Local Warm Cache

The P2P cache is effective once the cluster is warm, but the first wave faces the bootstrap paradox of §2.5 because no peers hold any data. CacheFS breaks the paradox with *CacheRoot*, a two-tier architecture that interposes a persistent, pre-warmed cache cluster between user instances and the remote storage backend (the fourth tier in the lookup chain, Figure 3).

A CacheRoot cluster comprises always-running nodes with large caches pre-populated with popular models. They expose a gRPC tracker service that, given `(target, block)`, returns addresses of nodes holding the block. Results are cached locally (5s TTL). Data transfer reuses coro-rpc. A dedicated NIC may be bound to CacheRoot traffic to isolate it from intra-cluster P2P. In production, we deploy one CacheRoot cluster per region. Once a user cluster is warm, reads are served by intra-cluster peers and CacheRoot traffic drops to zero. CacheRoot therefore absorbs only the initial cold-start burst.

3.7 Correctness and Robustness

Despite serving immutable artifacts, CacheFS must still ensure every consumer sees a self-consistent model view and that failures degrade cleanly. A mount caches namespace metadata with directory revalidation disabled, and each cache key encodes path, length, and `mtime`. Source files must remain immutable for the lifetime of a mount. A new model version is exposed through a new mount, which reads fresh metadata and derives new cache keys when the path, length, or `mtime` changes. Existing mounts do not automatically detect in-place updates. Network-transferred blocks are CRC32-verified before cache insertion. Gossip ownership is a performance hint rather than a correctness requirement. On a miss in source-backed mode, CacheFS tries ten peer candidates (30s timeout each), blacklists failures for 15s, then falls through to seeders, CacheRoot, and finally the source. In sourceless mode, a read returns an error if the requested block cannot be retrieved from the cooperative cache, without falling back to upstream storage (§3.2). CacheRoot tracker queries use a 2s per-address timeout with 5s result caching. Security and remaining robustness limitations are discussed in §6.3.

4 Implementation

CacheFS is implemented primarily in Go, with the high-throughput data transfer path written in C++ and accessed via CGo. The system comprises five major packages: the cache store (block management, local and remote caching, eviction), the metadata engine (read-only inode tree, sourceless provider/consumer logic), the Serf cluster module (membership, gossip events, consistent hash ring), the transport layer (coro-rpc bindings, RDMA/TCP, bandwidth control), and the daemon/SDK subsystem. CacheFS mounts via FUSE [30] using the `go-fuse` library. Because the file system is strictly read-only, the VFS layer implements only the read path (`Open`, `Read`, `Readdir`, `Lookup`, `GetAttr`, and `ReadLink`) and returns `EROFS` for all write operations. This reduces the implementation surface compared to a full-featured distributed file system.

4.1 Daemon Architecture

When multiple inference containers run on the same host, separate FUSE mounts would duplicate cache memory and Serf membership. CacheFS uses a *daemon architecture* in which a long-running daemon holds the cache store, Serf membership, and transport connections, while lightweight FUSE clients connect to it via UDS (or VSock for VM clients). The daemon provides *zero-copy block delivery* via `memfd`. It allocates blocks through `memfd_create()` and passes the file descriptor over UDS using `SCM_RIGHTS`. The client then maps the block directly with `mmap`, avoiding data copies between address spaces. This is critical when many containers simultaneously load the same model through a shared daemon.

4.2 Native SDK

Each FUSE read traverses the kernel VFS, crosses into user space via the FUSE device, and returns through the same path, incurring at least two context switches per operation. For workloads that are sensitive to per-read latency (rather than aggregate bandwidth), CacheFS provides native C and Python SDKs that communicate directly with the daemon process, bypassing the FUSE layer entirely.

Through the SDK's block-oriented API, applications open a target file by path, then read blocks by offset and length. Internally, the SDK issues the same daemon requests as the FUSE client but avoids the kernel round-trips. On a 40 GiB test file read sequentially from a warm cache, the SDK achieves 18.4 GiB/s, within 5% of local `tmpfs` (19.3 GiB/s) and over 3× the throughput of the FUSE path (5.9 GiB/s). Production retains FUSE because vLLM, SGLang, and Transformers already load model weights through file paths. The SDK serves integrations that can issue path, offset, and length requests directly.

4.3 Memory Management and RDMA

CacheFS manages hundreds of GB of cached data alongside a Go runtime whose GC is sized for much smaller heaps. All cache pages are allocated off-heap via `memfd_create()` + `mmap` (invisible to GC), with `madvise(MADV_HUGEPAGE)` to encourage 2 MiB pages and reduce TLB pressure on large

eviction scans. Pages use `Acquire`/`Release` reference counting and return to a reuse pool at zero count. To avoid redundant upstream fetches when multiple threads request the same block concurrently, a `PendingOpsManager` deduplicates in-flight requests. Only the first caller fetches the block, while the others wait for its result. We raise Go's soft memory limit to 1 GiB so GC cycles are infrequent.

The RDMA path (RoCE v2 over standard Ethernet) registers a pool of fixed-size 128 KiB bounce buffers at startup and pipelines transfers in chunks. RDMA capability is probed per connection. Failures fall back transparently to TCP. CacheRoot transport may be bound to a dedicated NIC to isolate cold-start bursts from intra-cluster P2P.

4.4 Implementation Complexity

CacheFS comprises ~30k lines of Go (production code, excluding tests, vendored libraries, and protobuf-generated stubs) and a ~500-line C++ shim bridging coroutine RPC via CGo, with the cache store (9.3k), VFS (5.6k), metadata engine (2.6k), Serf cluster (1.3k), and transport (0.7k+0.3k) forming the core.

All VFS write entry points return `EROFS` immediately, eliminating subsystems that a read-write file system requires: journaling, transaction coordinators, write-coalescing buffers, dirty-page tracking, copy-on-write block mapping, and metadata transactions. This keeps the implementation focused on the read-only model-loading path rather than general file-system updates. The metadata engine implements only `Lookup`, `ReadDir`, `GetAttr`, `ReadLink`, and `LoadMeta`, with no inode allocation, parent-pointer maintenance, quota, or ACL/xattr. The central new mechanism, hash-aware cooperative eviction, requires roughly 200 lines.

5 Evaluation

We evaluate CacheFS along eight dimensions: (1) end-to-end model distribution (§5.2); (2) CacheRoot cold-start acceleration (§5.3); (3) scalability with burst size (§5.4); (4) data-plane throughput and access-interface overhead (§5.5); (5) the proactive preload mechanism (§5.6); (6) per-instance latency distribution (§5.7); (7) hash-aware eviction under cache pressure (§5.8); and (8) ablation and production deployment experience (§5.9, §5.10).

5.1 Experimental Setup

Hardware and software. Production loads (§5.2, §5.3) run on GPU inference servers across three vendor GPU classes (a prior-generation 8×GPU server, a current-generation 8×GPU server, and an extended-memory variant). Scalability microbenchmarks (§5.4) use general-purpose VMs at three NIC tiers: `Node-10G`, `Node-16G`, and `Node-25G`. `Node-16G` is the cloud's standard high-bandwidth tier with 16 Gbps steady and 32 Gbps burst. The transport microbenchmarks (§5.5) use 200 Gbps NICs for TCP and 400 Gbps NICs for RDMA (RoCE v2). The production storage configurations provide 1–5 Gbps per client. All experiments run on Linux. Production services use Kubernetes orchestration, with vLLM and SGLang as inference engines. CacheFS uses 128 MiB blocks in production and in the model-file experiments. Other defaults are a 128 GiB in-memory cache, 24-block prefetch, hash-aware eviction, 100 virtual nodes per physical node, 200ms gossip interval, 30s peer timeout, and 15s blacklist. The file-read experiments in §5.6–5.8 use `safetensors` on general-purpose VMs (96 vCPU, 384 GiB RAM, 32 Gbps baseline and 48 Gbps burst bandwidth) to reproduce SGLang's model-loading I/O without requiring GPUs.

Models and baseline. We evaluate seven production models ranging from 62 GB to 643 GB. They are Qwen2.5-32B and Qwen2.5-72B, Qwen3-32B and Qwen3-235B [41], MiniMax-M2, DeepSeek-R1, and DeepSeek-V3.2 [13]. Except for the Qwen2.5-72B 50-instance result, whose baseline is cloud NFS, all other experiments that access upstream storage use cloud object storage as the baseline or source.

5.2 End-to-End Model Distribution

Figure 4 summarizes end-to-end model distribution for three production models, all measured during real inference scale-out. The plotted values are from one production scale-out event per model. The microbenchmarks in §5.4, §5.6, and §5.8 are averaged over at least three runs and report distributions.

Scale-out results. For Qwen2.5-72B with cloud NFS, all 50 instances contend for upstream bandwidth, requiring ~28 minutes for the batch. With CacheFS, a few provider nodes seed the model, and subsequent instances fetch from peers. The 50-instance scale-out completes in 30 seconds, a **56**× speedup. The P2P path replaces NFS's $O(N)$ contention with $O(\log N)$ propagation. With cloud object storage, Qwen2.5-72B requires 787s per instance at N=10 and 6,666s per instance at N=100. CacheFS reduces the N=100 per-instance read time to ~150s (§5.6), a **44**× speedup. Aggregate throughput remains nearly flat at 13.8 Gbps and 16.3 Gbps as concurrency grows tenfold, indicating a shared upstream bottleneck rather than a client-NIC limit. The warm-path NIC-scaling result in §5.9 independently confirms that client NIC bandwidth is not the ceiling. We did not isolate the storage, gateway, and network contributions. If upstream storage already meets the burst-readiness target, CacheFS provides less benefit on the cold path. Warm peer-served waves still reduce backsource to approximately 0.3% or less (§5.4).

Per-node results. DeepSeek-R1 per-node load drops from ~19 minutes to 200s (**82.5%** reduction). The remainder is dominated by host-to-device transfer and framework init, confirming the storage bottleneck is gone. For Qwen3-235B (~470 GB), per-node startup drops from 6 minutes to 60s (**6**×). End-to-end startup (with container scheduling, image pull, and runtime init) drops ~9.4 minutes→~4.3 minutes. Both

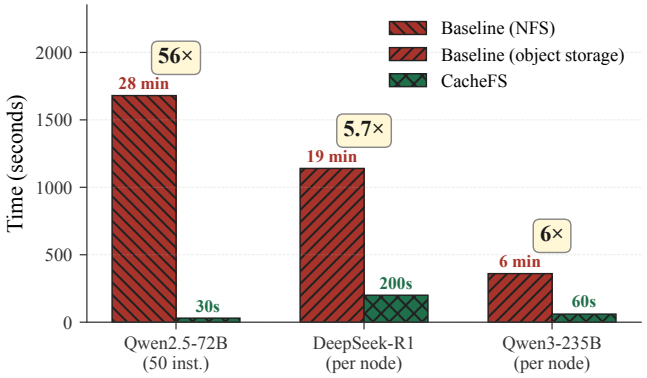


Figure 4. End-to-end model distribution time. The Qwen2.5-72B 50-instance baseline uses cloud NFS, while the DeepSeek-R1 and Qwen3-235B per-node baselines use cloud object storage. The measured improvements are 56×, 5.7× (82.5% reduction), and 6×, respectively.

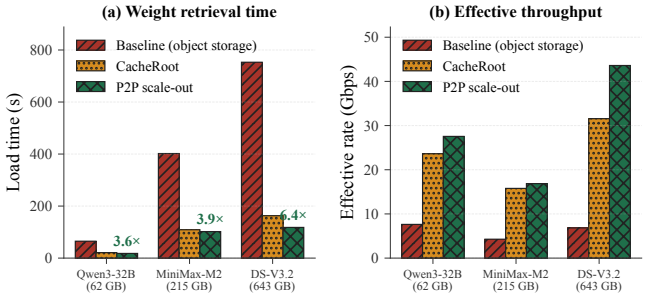


Figure 5. CacheRoot and P2P acceleration across three production models. The direct baseline in both panels is cloud object storage. (a) P2P improves weight retrieval time by 3.6–6.4×. (b) P2P reaches up to 43.6 Gbps for DeepSeek-V3.2.

the 6-minute baseline and the 60-second result include framework initialization. The 6× improvement therefore combines CacheFS's weight-retrieval reduction (~5.5 minutes→~45s) with the companion runtime's reduction of framework initialization. Other startup phases account for most of the remaining time.

5.3 CacheRoot vs. P2P Acceleration in Production

We isolate CacheRoot's contribution from intra-cluster P2P by measuring two distinct production scale-out phases on three models. *Cold-start (CacheRoot)*: a fresh deployment where the user-cluster cache is empty and all blocks must be fetched from the pre-warmed CacheRoot tier. *Scale-out (P2P)*: a subsequent burst into a cluster whose prior wave has already warmed the P2P cache, so blocks are served entirely by intra-cluster peers. Figure 5 summarizes the results.

CacheRoot reduces cold-start weight retrieval time by 3.1–4.6× across the three models. Subsequent P2P-served scale-out further improves to 3.6–6.4× and reaches up to

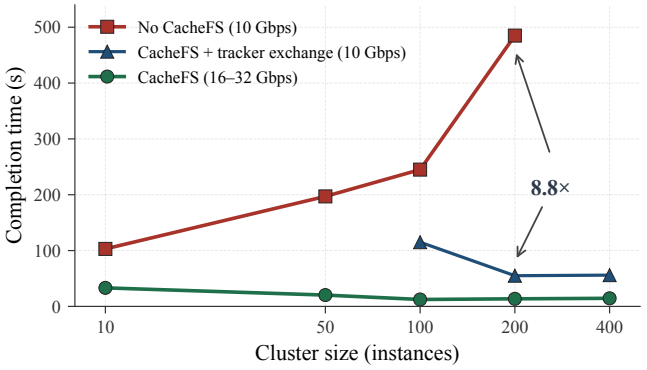


Figure 6. Scale-out completion time vs. burst size. Without CacheFS, time grows with cluster size as instances contend for shared upstream bandwidth. With CacheFS, once the P2P cache is warm (after the first wave), completion time remains near-constant regardless of burst size. High-bandwidth NICs (Node-16G, 16–32 Gbps) further reduce absolute time compared to 10 Gbps nodes.

43.6 Gbps for DeepSeek-V3.2. For Qwen3-32B, E2E service readiness falls from 103s to 58s. For models that exceed the in-memory cache quota, a disk-backed cache variant trades throughput for capacity (13.9–15.6 Gbps) while remaining well above the baseline.

5.4 Scalability with Burst Size

We test how CacheFS scales with burst size by issuing progressively larger waves of Qwen2.5-32B (63 GB, 30 files). The first wave populates the cooperative cache from upstream. Subsequent waves are served primarily by the warm P2P layer. We report per-wave completion time (the mean elapsed time for newly added instances to finish loading) and the fraction of data served from peers vs. source (Figure 6).

High-bandwidth (Node-16G, 16–32 Gbps). The first 10-instance wave has a mean completion time of 33s and mean upstream reads of 9.7 GiB/instance. Once warm, subsequent bursts complete in 12–15s regardless of N: 50→100 in 12.4s, 100→200 in 13.6s, and 200→400 in 14.5s with approximately 0.3% backsource. Completion time is effectively *independent of N* once the cooperative cache warms.

Moderate-bandwidth (Node-10G, 10 Gbps). With gRPC tracker exchange (§3.3), CacheFS achieves 55s for 100→200 and 56s for 200→400. The 200→400 wave records less than 0.3% backsource. Without tracker exchange, 100→200 takes 83s with mean upstream reads of 2.1 GiB/instance. For the 100→200 wave, mean completion time falls from 485s without CacheFS to 55s with CacheFS and tracker exchange, a **8.8**× speedup (Figure 6). Warm-cluster waves fetch nearly all data from peers. gRPC tracker exchange reduces completion time by 34% at large N. Completion time also scales

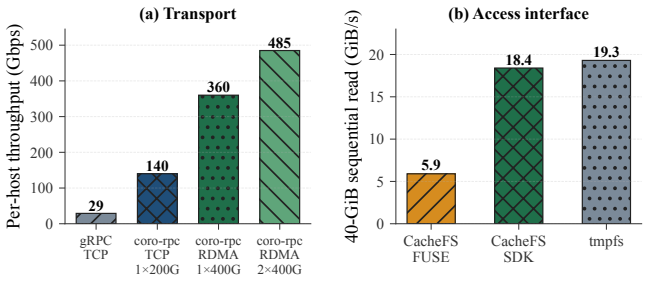


Figure 7. (a) Per-host data-plane throughput under multi-connection aggregate workloads. coro-rpc/TCP (140 Gbps, model-distribution path) achieves 4.8× gRPC throughput. Optional coro-rpc/RDMA reaches 360 Gbps on a single 400 Gbps NIC and 485 Gbps as a two-NIC aggregate for fast-startup scenarios. (b) Access interfaces on a 40 GiB warm-cache read. The SDK achieves 3.1× FUSE throughput and is within 5% of `tmpfs`. FUSE remains the model-distribution path and is not the bottleneck for bandwidth-bound weight loads.

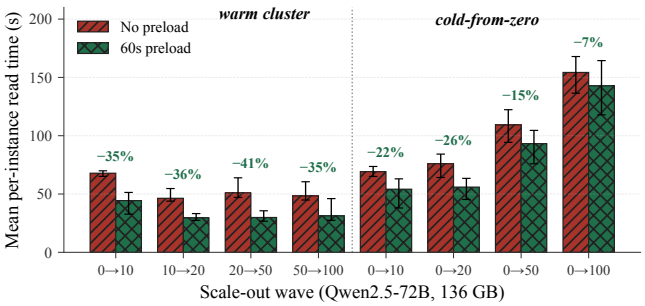


Figure 8. Effect of a 60s proactive preload window on per-instance read time across eight scale-out waves of Qwen2.5-72B. Preload reduces read time by 7–41% (mean 27%). Extending the window to 200s on the warm 10 → 100 wave further drops read time from 128.4s to 29.5s (measured separately).

with available NIC bandwidth (55s at 10 Gbps vs. 14.5s at 16–32 Gbps).

5.5 Throughput and Access Interface

Figure 7(a) reports per-host data-plane throughput under a multi-connection aggregate workload. The TCP benchmarks use hosts with 200 Gbps NICs, while the RDMA benchmarks use RoCE v2-capable hosts with 400 Gbps NICs. Over TCP, coro-rpc reaches 140 Gbps on a single 200 Gbps NIC. This is **4.8**× gRPC throughput, due to avoiding Protobuf serialization and using coroutine scheduling. This is the path used by the production workload of §5.2–5.8. Over RDMA, coro-rpc sustains 360 Gbps on a single 400 Gbps NIC and 485 Gbps with two such NICs aggregated. Beyond two NICs, aggregate throughput plateaus (likely due to memory pages not being pre-faulted) and remains open work. We use the RDMA path for latency-critical fast-startup scenarios.

Figure 7(b) reports access-interface throughput on a 40 GiB sequential warm-cache read. The SDK achieves 18.4 GiB/s, which is **3.1**× FUSE throughput and within 5% of `tmpfs`. It bypasses the FUSE kernel–userspace round-trip via direct daemon communication and `memfd`-based zero-copy (§4.1). The SDK is used by the companion fast-startup runtime that is read-overhead-sensitive. The production model-distribution path uses FUSE for transparent POSIX compatibility and is bandwidth-bound.

5.6 Proactive Preload

We next quantify the proactive preload mechanism described in §3.4. In production, vLLM and SGLang spend 30–60s on container start, framework initialization, and shard planning before they read the first weight byte. Preload converts that idle window into useful background work by fetching the target model into the local cache (and, with `--broadcast`, into the entire Serf cluster) ahead of demand. We replicate this scenario by running the same model-loader that vLLM/SGLang invoke (`safetensors`) after a configurable preload sleep.

Setup. We load Qwen2.5-72B (136 GB, 37 safetensor files) on general-purpose VMs (96 vCPU, 384 GiB RAM, 32 Gbps baseline and 48 Gbps burst bandwidth). We use 128 MiB blocks, 24-block read-ahead, and a 128 GiB in-memory cache. GPUs are not required because the test only exercises file reads. A configurable sleep emulates the framework initialization window. We run the scale-out sequence [10, 20, 50, 100, 0, 10, 0, 20, 0, 50, 0, 100] (the second half drops to zero between waves to test cold-from-zero behavior), reporting per-instance mean read time with and without a 60s preload window.

Results. Figure 8 shows preload reduces read time by 7–41% across all eight waves (mean **27%**). On *warm-cluster* waves (0 → 10 → 20 → 50 → 100), the reduction is a consistent 35–41% from overlapping P2P fetches with framework init. On *cold-from-zero* waves, the cluster is reset to empty before each burst, so preload fetches from upstream into an empty cluster. The gain shrinks with N as the 60s seed window covers a smaller fraction of total bytes (26% at N=20 falling to 7% at N=100). A longer preload window further reduces read time. With a 200s preload, the warm 10 → 100 wave drops from 128.4s to 29.5s (measured separately). The benefit comes from overlapping weight retrieval with framework initialization and from eliminating P2P/backsource hops at read time.

5.7 Tail Latency

Mean read time hides cold-start tail behavior, which matters when each pod becomes ready independently. Figure 9 reports min/P50/P90/max per-instance read time for the same 8-wave run. Warm-cluster waves are tightly clustered (P90/min<1.2, max/min<1.4). On the 50 → 100 wave, P90 is 31.1s, 1.07× the mean. Cold-from-zero waves widen with N (min–max spread grows from 21s at N=10 to 53s at N=100),

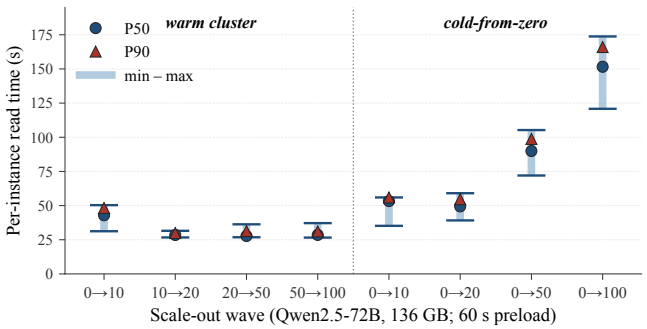


Figure 9. Per-instance read time distribution (min, P50, P90, and max) across eight Qwen2.5-72B scale-out waves (60s preload). Warm waves (10 → 20 → . . . → 50 → 100) are tightly clustered. Cold-from-zero waves widen with N but exhibit no long tail (P90 remains within 5% of the maximum at N=100).

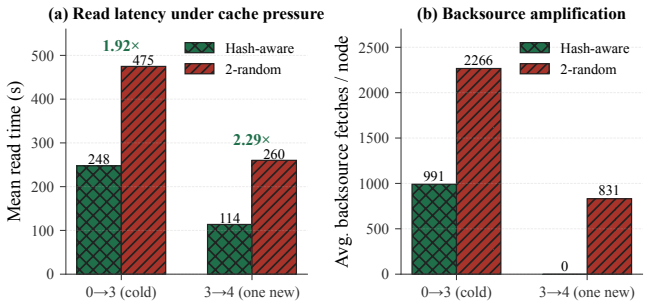


Figure 10. Hash-aware vs. 2-random eviction under cache pressure (60 GiB cache, 136 GB Qwen2.5-72B working set). (a) Hash-aware halves read time on the cold 0 → 3 burst and is 2.3× faster on the 3 → 4 wave. (b) Backsource amplification. Hash-aware drives the 3 → 4 wave to zero upstream fetches. The cluster's combined cache covers the entire model.

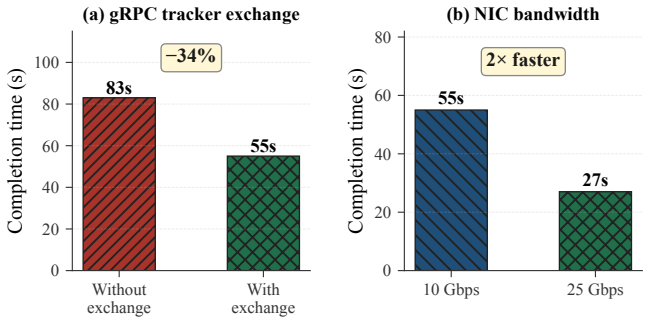


Figure 11. Ablation results. **Left:** gRPC tracker exchange reduces 100→200 scale-out time by 34% (83s → 55s). **Right:** With tracker exchange enabled, increasing NIC bandwidth from 10 to 25 Gbps reduces completion time from 55s to 27s.

but exhibit no long tail. At N=100, P90 is about 10% above the mean and within 5% of the maximum. Epidemic distribution bounds the slowest instance close to the median.

5.8 Hash-aware Eviction under Cache Pressure

A consumer cache that is smaller than the working set must evict. The question is which blocks to remove. CacheFS's hash-aware eviction (§3.4) deliberately retains blocks for which the local node is the consistent-hash owner and discards the rest, so that the union of caches across the cluster converges toward one copy of every block. We compare it against 2-random, a classical policy that selects two candidates uniformly at random and evicts the older.

Setup. Per-node cache is limited to 60 GiB, less than half the 136 GB working set. Every node must therefore evict. We exercise two patterns: a cold 0 → 3 burst and a 3 → 4 burst that admits a fourth node into a cluster where the prior three have evicted to fit. We report mean per-instance read time and the average number of backsource fetches per node. This is a policy-to-policy comparison under identical cluster and workload conditions.

Results. As Figure 10 shows, hash-aware eviction halves read time in the cold burst (247.6 vs. 474.8s, 1.92×) and cuts backsource amplification from 2,266 to 991 blocks per node. The difference is larger on the 3 → 4 wave. With hash-aware eviction, the three earlier nodes collectively hold every block, so the new node fetches entirely from peers (**0 backsource**, 113.7s), while 2-random's overlapping evictions leave some blocks absent from all peers, forcing 831 backsource fetches (260.2s, −56%). Retaining different blocks on different nodes allows the union of local caches to behave like one cluster-sized cache, even when no individual node holds the model.

5.9 Ablation and Sensitivity

gRPC tracker exchange. On `Node-10G` with a 100→200 burst, we compare the Serf-based tracker bootstrap with gRPC tracker exchange (§3.3). Without it, mean completion time is 83s with mean backsource reads of 2.1 GiB/instance. With the exchange, mean completion time drops to 55s (**34%** reduction) with zero backsource (Figure 11 left). The improvement comes from obtaining the initial tracker view directly from existing peers, avoiding incomplete tracker views during the join burst.

NIC bandwidth sensitivity. With tracker exchange enabled, raising NIC bandwidth from 10 to 25 Gbps (2.5×) reduces 100→200 completion from 55s to 27s (approximately 2×), broadly tracking the increase in available bandwidth (Figure 11 right).

CacheRoot ablation. Figure 5 shows that CacheRoot improves first-wave cold-start by 3.1–4.6× across all three production models. Once warm, subsequent scale-out bypasses CacheRoot entirely and reaches up to 43.6 Gbps.

Control-plane scaling. On Node-16G, mean completion time increases only slightly from 13.6s at 100→200 to 14.5s at 200→400. Backsource at 400 nodes is approximately 0.3%, consistent with fast metaMap convergence.

5.10 Production Deployment

CacheFS has been in production for >18 months, serving a fleet at the hundred-thousand-GPU scale across four regions and covering a majority of the platform's inference GPUs. The production fleet uses both cloud NFS and object storage as upstream backends. The results in this subsection are production observations rather than controlled A/B measurements.

Deployment timeline. CacheFS went live in November 2024 on a single-tenant deployment, expanding within two months to a multi-tenant fleet at the tens-of-thousands-of-GPUs scale. Hash-aware eviction was enabled by default from the first release. CacheRoot was introduced in September 2025 in response to a cloud public-model registry whose cold-start batches pulling shared public models from object storage saturated upstream egress. We now deploy one CacheRoot cluster per region across four cloud regions.

GPU-hour savings. The system saves more than **7,700** GPU card-hours per day by reducing warm-standby pools. Previously, pools were sized to the peak burst (30–40% of peak instance count). CacheFS reduces wave completion below the autoscaler interval, allowing pools to be sized to average demand. It removes the dominant storage bottleneck, while a companion fast-startup runtime (outside the scope of this paper) reduces framework initialization. The combined effect enables the pool reduction. The 7,700 card-hours represent one day on a current-generation GPU fleet (8 cards/instance, spanning Qwen2.5/3 and DeepSeek-R1/V3).

Operational notes. Four observations from the deployment have shaped how we operate CacheFS at scale. *(i) The first wave still matters.* Even with CacheRoot pre-warming, the very first batch of a new model behaves measurably worse than later waves. We now schedule a small *priming wave* (3–5 instances) ahead of expected demand whenever a new model is published, so the user-visible first batch never hits a fully cold cluster. *(ii) Cache pressure has shifted from rare to dominant.* When most models were under ∼50 GB, fewer than 5% of pod-hours ran with $NC < B$, and hash-aware eviction (§5.8) was exercised only on the tail. As model sizes have grown, this fraction now exceeds 60%, and hash-aware eviction has become the cluster's most-exercised mechanism. *(iii) Cross-version heterogeneity surfaces at the CacheRoot boundary, not within deployments.* Per-deployment isolation keeps each user cluster on a single CacheFS version, but user clusters routinely talk to a long-lived CacheRoot tier that may run an older or newer build. We rely on Protobuf's natural backward/forward compatibility (unknown fields are ignored) and use only additive schema changes. This is sufficient to let mixed-version clusters coexist without coordinated migration. *(iv) CacheFS clusters are per-deployment, not fleet-wide.* Separate Serf groups provide logical deployment separation. Security relies on infrastructure-level isolation (§6.3).

Table 1. Summary of key evaluation results.

Result	Improvement	§
Qwen2.5-72B, 50-inst. vs. NFS	56× (28 min→30s)	5.2
Qwen2.5-72B, 100-inst. vs. obj. storage	44× (111 min→150s)	5.2
DeepSeek-R1, per-inst. vs. obj. storage	82.5% (19 min→200s)	5.2
Qwen3-235B, per-inst. vs. obj. storage[†]	6× (360s→60s)	5.2
DS-V3.2, P2P rate vs. obj. storage	43.6 Gbps (6.4×)	5.3
coro-rpc TCP vs. gRPC	4.8× (29→140 Gbps)	5.5
SDK vs. FUSE	3.1× (5.9→18.4 GiB/s)	5.5
60s preload (warm)	−35–41%	5.6
Hash-aware vs. 2-random (3→4)	0 vs. 831 backsource/node	5.8
GPU-hours saved per day	7,700+	5.10

[†] With the companion fast-startup runtime.

Table 1 consolidates the key results.

6 Production Experience and Lessons

6.1 Lessons from Production

Gossip alone is not a reliable substrate for high-volume state convergence. CacheFS uses Serf User Events for runtime block-ownership announce/revoke. Its original join-time bootstrap used a CacheFS-defined dump event over Serf. In CacheRoot, heavier cross-deployment traffic left joiners with incomplete tracker views, causing unnecessary backsource reads. We replaced this bootstrap with gRPC tracker exchange (§3.3) for both large startup bursts and CacheRoot joins. Each joiner exchanges tracker state with existing peers and periodically reconciles with a random peer (default interval 120s, configurable). During large startup bursts, Go GC overhead also reduced the network throughput of gRPC tracker exchange. We raised Go's soft memory limit to 1 GiB to reduce GC frequency (§4.3). Runtime announce/revoke remains on Serf. Gossip suits steady-state membership and incremental events, while full-state bootstrap benefits from a direct request-response path that does not compete with the event stream.

Preload trumped read-ahead. The original prefetch design carefully tuned read-ahead window depth. In production, the most impactful intervention turned out not to be read-driven prefetch but *proactive preload* during framework init. vLLM and SGLang spend 30–60s on container start, CUDA initialization, and shard planning before the first weight byte is requested. Preload uses this idle window to warm the local cache. Whether it can cache the entire model depends on cache capacity, available throughput, and the window's duration. A single `--broadcast` flag fans the

warm-up to every node in the cluster in one RPC. The lesson is to look for application-level idle windows rather than tuning the data path further.

The OS OOM killer is the dominant failure mode. Our initial implementation allocated cache buffers on the Go heap. With the default `GOGC=100`, Go's heap target is approximately twice the live heap. Delayed reclamation of evicted buffers could therefore cause memory consumption to approach twice the configured cache size, resulting in OOM-induced restarts. We moved cache pages off-heap via `memfd_create()` and `mmap`, and managed their lifetime with explicit reference counting (§4.3), separating cache memory management from GC scheduling.

Co-located inference workloads can still exhaust host memory. Because CacheFS uses otherwise idle RAM and is often the largest non-kubelet process on a node, it is a likely OOM victim. Production therefore sizes the cache as a fraction of host RAM, subject to a minimum cache size, and adjusts the daemon's OOM `score_adj` relative to inference containers and critical system processes. The evaluation uses fixed cache sizes.

6.2 Why Not Existing Systems?

From an operational perspective, we considered and rejected three alternatives. **3FS** [14] was not open-sourced until February 2025, after CacheFS shipped in November 2024. As a full read-write file system, it includes chain replication and FoundationDB metadata that our read-only workload does not require. The **Alluxio** design [27] uses a centralized master for metadata management. **Dragonfly** [8] requires centralized schedulers and a Redis state store [40], exposes no POSIX interface, and targets layered OCI images rather than hundreds-of-GB weight blobs. CacheFS instead uses gossip membership and consistent-hash placement. Its tracker and data path run within each deployment, without a separate master, scheduler, ZooKeeper, or Redis service.

6.3 Limitations and Threats to Validity

CacheFS uses plaintext intra-cluster transport and relies on infrastructure-level isolation (VPC, namespace scoping). Each deployment runs an isolated Serf group (§5.10), providing logical separation between deployments. However, *within* a single CacheFS cluster there is no data-plane authentication. Serf group tags prevent accidental cross-cluster mixing but are not adversary-grade. mTLS and attested Serf joins are planned for deployments requiring intra-cluster tenant isolation. Our quantitative measurements use cloud NFS and object storage because these are the production platform's storage backends. A same-hardware head-to-head comparison with 3FS or Dragonfly remains open work. The reported end-to-end gains therefore reflect both cooperative P2P caching and the configured upstream paths. The evaluation does not separate their individual contributions. We have not evaluated CacheFS on general VM or container content-distribution workloads. CacheFS does not provide cross-node single-flight for upstream misses. Multiple nodes may fetch the same block before gossip propagates the new ownership announcement. Graceful-degradation behavior is observational, not from controlled fault injection. CacheFS is tuned for read-only, full-file, homogeneous reads. Workloads with fragmented or random-access patterns (e.g., LoRA adapters) fall outside our current policies.

6.4 Open Directions

Four extensions are under exploration. *Mutable artifacts*: a generation-tagged `(target, gen)` key extends immutable-snapshot metadata to KV-cache state (shared across requests or nodes for prefix reuse) and fine-tune deltas at the cost of a per-generation miss. *Scheduler-integrated preload*: issuing preload at K8s admission time rather than from the framework's init script would overlap preload with image pull and further reduce the remaining cold-start window. *GPUDirect RDMA*: streaming blocks directly into GPU memory removes the PCIe host-to-device hop for the largest models. *Memory-resource coordination with KV-cache systems*: CacheFS and KV-cache offloading systems (e.g., Mooncake [33], vLLM CPU offload) both target host idle memory. CacheFS and KV-cache offloading currently set their memory limits independently even though both draw from the same host-DRAM pool. A coordinated allocator could shift memory between model weights and KV state as scale-out demand and active context length change, reducing contention and OOM events.

7 Related Work

General-purpose distributed file systems. Ceph [39], Lustre [35], GPFS [34], HDFS [36], and JuiceFS [23] provide read-write semantics via centralized metadata services optimized for mixed workloads. CacheFS specializes consumer-side cache cooperation for burst weight loading, allowing consumers to serve cached blocks to peers.

AI-oriented storage. 3FS [14] is a high-throughput distributed file system with chain replication, FoundationDB metadata, and an RDMA data plane for AI training/inference. 3FS is optimized for RDMA networks, whereas CacheFS supports TCP-only deployments with optional RDMA acceleration. 3FS's reported per-client peak is ≈320 Gbps on InfiniBand. The key architectural difference is that 3FS includes a write path, replication protocol, and metadata service that our read-only workload does not require, even if it could be co-deployed on inference hosts. CacheFS assumes no storage ownership. Training, model updates, and durable storage remain upstream, while cache replication follows inference-replica homogeneity. The two systems are complementary. CacheFS reuses memory on inference hosts for

model distribution, whereas 3FS provides a read-write storage system. Tensorizer [10] accelerates per-process tensor loading but does not share state across nodes. Quiver [25] and Fluid [19] apply training-aware caching to data lakes. Their random-access shuffled-sample working sets do not transfer to inference cold-start.

LLM cold-start and serving. Recent LLM systems optimize different stages of startup and serving: Serverless-LLM [16] reduces cold-start via fast multi-tier checkpoint loaders; vLLM [26] and SGLang [43] manage KV-cache memory; DistServe [44] and Splitwise [32] disaggregate prefill and decode; Llumnix [37] live-migrates requests; AlpaServe [29] multiplexes model parallelism. CacheFS accelerates weight delivery to an instance and complements these serving systems. CachedAttention [17], Mooncake [33], and LMCache [7] cache KV-cache state rather than model weights. We discuss resource coordination with these systems in §6.4.

P2P content distribution. Dragonfly [8] and Kraken [38] accelerate container-image distribution at the HTTP layer with centralized schedulers, in the spirit of BitTorrent's swarm protocol [9], but neither exposes POSIX. Adjacent work includes Slacker [20], DADI [28], Starlight [6], and analyses of registry workloads [2]. These systems accelerate container-image pull by reorganizing image layout or enabling lazy block-level fetches, but inherit the layered/OCI semantics rather than the single-file, gigabyte-scale weight-blob shape that CacheFS specializes for. A recent TON paper [40] extends Dragonfly with a graph-learning-based scheduler for large-model distribution, but its centralized scheduler + Redis architecture is the pattern CacheFS avoids, and its testbed operates at ∼20× lower per-node bandwidth with ∼14× smaller artifacts than our environment. These systems can distribute model bytes. CacheFS addresses the need to load all weight files before serving requests and reuses data among replicas in the same scale-out burst (§2.3).

Distributed caching, gossip, consistent hashing. Cooperative client-memory caching has a long lineage. xFS's serverless cache forwarding [1] and Dahlin et al.'s cooperative caching study [11] both treat peer RAM as a tier between local cache and disk. These designs assume a tightly-coupled cluster, a centralized metadata authority, or a writable workload, none of which match the read-only, gossip-only LLM weight-loading regime. The Alluxio design [27] provides memory-speed access with master-managed metadata. Memcached [31] exposes key–value objects, but does not provide file semantics or coordinate retention across nodes. CacheLib [3] is an in-process engine. RobinHood [4] reallocates cache capacity across backends for tail-latency reduction at a different layer. With hash-aware cooperative eviction, nodes use consistent hashing to select blocks to retain, allowing the cluster to approximate a single non-redundant cache without centralized metadata. Unlike Dynamo-style systems and consistent-hash load balancers (e.g., Maglev [15]) that use the hash ring to *route*, CacheFS uses it only for *retention*, while routing is data-driven over gossip [12, 21, 24], so the cooperative cache is resilient to ring rebalancing without coupling correctness to ring agreement.

8 Conclusion

CacheFS has operated for more than eighteen months across four cloud regions on a fleet at the hundred-thousand-GPU scale. Together with a companion fast-startup runtime, it saves more than 7,700 GPU card-hours per day by reducing warm-standby pools. The design combines hash-aware cooperative eviction with sourceless P2P distribution, gossip coordination, proactive preload, a coroutine-based TCP data plane, optional RDMA, and a warm CacheRoot tier. These mechanisms follow from the narrow LLM weight-loading workload and the operational constraints observed in production. Our production experience provides lessons on cache coordination, preload timing, managed-runtime memory, and fleet-scale operation.

Acknowledgments

We thank the anonymous reviewers, our shepherd, and the artifact evaluators for their valuable feedback that improved this paper. This work was supported by the National Natural Science Foundation of China (Grant No. 62302441), Alibaba Innovative Research Program, Shandong Provincial Higher Education Institutions Youth Innovation Team Development Plan (No. 2025KJH023), and Zhejiang University Education Foundation Qizhen Scholar Foundation.

Artifact Availability

The evaluated artifact (v1.0.0) is archived on Zenodo at https://doi.org/10.5281/zenodo.23167350.

References

[1] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. 1995. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Copper Mountain, CO, USA, 109–126. doi:10.1145/224056.224066

[2] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. 2018. Improving Docker Registry Design Based on Production Workload Analysis. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Oakland, CA, USA, 265–278.

[3] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Virtual, 753–768.

[4] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail Latency Aware Caching—Dynamic Reallocation from Cache-Rich to Cache-Poor. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Carlsbad, CA, USA, 195–212.

[5] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade. In *Communications of the ACM*, Vol. 59. ACM, 50–57. doi:10.1145/2890784

[6] Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara. 2022. Starlight: Fast Container Provisioning on the Edge and over the WAN. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Renton, WA, USA, 35–50.

[7] Yihua Cheng, Kuntai Du, Jiayi Yao, and Junchen Jiang. 2024. Do Large Language Models Need a Content Delivery Network? *arXiv preprint arXiv:2409.13761* (2024). doi:10.48550/arXiv.2409.13761

[8] CNCF Dragonfly. 2024. Dragonfly: An Intelligent P2P-Based Image and File Distribution System. https://d7y.io. Cloud Native Computing Foundation; accessed: 2026-05-15.

[9] Bram Cohen. 2003. Incentives Build Robustness in BitTorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*. Berkeley, CA, USA.

[10] CoreWeave. 2023. Tensorizer: Module, Model, and Tensor Serialization/Deserialization. https://github.com/coreweave/tensorizer. Accessed: 2026-05-15.

[11] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. 1994. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Monterey, CA, USA, 267–280.

[12] Abhinandan Das, Indranil Gupta, and Ashish Motivala. 2002. SWIM: Scalable Weakly-Consistent Infection-Style Process Group Membership Protocol. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, Bethesda, MD, USA, 303–312. doi:10.1109/DSN.2002.1028914

[13] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. https://arxiv.org/abs/2412.19437. arXiv:2412.19437; accessed: 2026-05-15.

[14] DeepSeek-AI. 2025. 3FS (Fire-Flyer File System): A High-Performance Distributed File System for AI Training and Inference. https://github.com/deepseek-ai/3FS. Open-sourced February 2025; system design described in [42]. Accessed: 2026-05-15.

[15] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Santa Clara, CA, USA, 523–535.

[16] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Santa Clara, CA, USA, 135–153.

[17] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC)*. USENIX Association, Santa Clara, CA, USA, 111–126.

[18] Google. 2016. gRPC: A High-Performance, Open-Source Universal RPC Framework. https://grpc.io. Accessed: 2026-05-15.

[19] Rong Gu, Kai Zhang, Zhihao Xu, Yang Che, Bin Fan, Haojun Hou, Haipeng Dai, Liyue Yi, Yu Ding, Yihua Huang, and Guihai Chen. 2022. Fluid: Dataset Abstraction and Elastic Acceleration for Cloud-Native Deep Learning Training Jobs. In *Proceedings of the 38th IEEE International Conference on Data Engineering (ICDE)*. IEEE, 2183–2196. doi:10.1109/ICDE53745.2022.00209

[20] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA, USA, 181–195.

[21] HashiCorp. 2013. Serf: Decentralized Cluster Membership, Failure Detection, and Orchestration. https://github.com/hashicorp/serf. Project site: https://www.serf.io; accessed: 2026-05-15.

[22] Hugging Face. 2023. Safetensors: Simple, Safe, and Fast Tensor Serialization. https://github.com/huggingface/safetensors. Accessed: 2026-05-15.

[23] JuiceData, Inc. 2021. JuiceFS: A POSIX-Compatible Shared File System for Cloud Environments. https://github.com/juicedata/juicefs. Accessed: 2026-05-15.

[24] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*. ACM, El Paso, TX, USA, 654–663. doi:10.1145/258533.258660

[25] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA, USA, 283–296.

[26] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Koblenz, Germany, 611–626. doi:10.1145/3600006.3613165

[27] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*. ACM, Seattle, WA, USA, 6:1–6:15. doi:10.1145/2670979.2670985 Now known as Alluxio.

[28] 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 Annual Technical Conference (ATC)*. USENIX Association, Virtual, 727–740.

[29] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Boston, MA, USA, 663–679.

[30] Linux Kernel Developers. 2024. FUSE: Filesystem in Userspace. https://github.com/libfuse/libfuse. Accessed: 2026-05-15.

[31] Memcached Project. 2011. Memcached: A Distributed Memory Object Caching System. https://memcached.org/. Accessed: 2026-10-04.

[32] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the 51st ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132. doi:10.1109/ISCA59077.2024.00019

[33] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation—A KVCache-Centric Architecture for Serving LLM Chatbot. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA, USA, 155–170.

[34] Frank Schmuck and Roger Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Monterey, CA, USA, 231–244.

[35] Philip Schwan. 2003. Lustre: Building a File System for 1000-Node Clusters. In *Proceedings of the 2003 Linux Symposium*. Ottawa, Canada, 380–386.

[36] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, Incline Village, NV, USA, 1–10. doi:10.1109/MSST.2010.5496972

[37] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Santa Clara, CA, USA, 173–191.

[38] Uber Technologies. 2019. Kraken: P2P Docker Registry Capable of Distributing TBs of Data in Seconds. https://github.com/uber/kraken. Accessed: 2026-05-15.

[39] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Seattle, WA, USA, 307–320.

[40] Zhou Xu, Lizhen Zhou, Zichuan Xu, Wenbo Qi, Jinjing Ma, Song Yan, Yuan Yang, Min Huang, Haomiao Jiang, Qiufen Xia, and Guowei Wu. 2026. Empowering Dragonfly: A Lightweight and Scalable Distribution System for Large Models with High Concurrency. *IEEE/ACM Transactions on Networking* 34 (2026), 339–354. doi:10.1109/TON.2025.3598067

[41] An Yang, Anfeng Li, Baosong Yang, et al. 2025. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388* (2025). doi:10.48550/arXiv.2505.09388

[42] Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Huazuo Gao, et al. 2025. Insights into DeepSeek-V3: Scaling Challenges and Reflections on Hardware for AI Architectures. In *Proceedings of the 52nd ACM/IEEE International Symposium on Computer Architecture (ISCA)*. ACM. doi:10.1145/3695053.3731412

[43] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Proceedings of the 38th Conference on Neural Information Processing Systems (NeurIPS)*. Vancouver, BC, Canada.

[44] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Santa Clara, CA, USA, 193–210.