Seekable OCI: Lazy-Loading Container Images via Range-Request Indexing

James Thompson, Jesse Butler, Sri Saran Balaji Rajakumar, Henry Wang
Amazon Web Services

***Abstract*—Container image pulling accounts for the majority of pod startup time in Kubernetes environments. Standard pull downloads the entire image before the container can start, even when the application accesses only a fraction of the image content at startup. We present SOCI (Seekable OCI), a lazy-loading architecture that enables containers to start without downloading the full image. SOCI builds an external index over standard OCI images, mapping files to byte ranges within compressed layers. At runtime, a FUSE filesystem intercepts file accesses and serves them via HTTP range requests. Unlike prior approaches that require image format conversion, SOCI works with unmodified images and standard registries. The index is stored as an OCI referrer artifact, requiring no changes to images, registries, or deployment tooling. On a 1.3 GB Python web service image, SOCI reduces cold-start pull time from 20 seconds to 2.5 seconds (7× speedup), with pull time independent of image size. We measure a crossover at 80% access density: below this, lazy loading wins; above, parallel full pull is faster. SOCI is deployed in production across Amazon EKS, Amazon ECS Fargate, and EKS Auto Mode, serving millions of image pulls daily.**

I. Introduction

Container image pulling is the dominant contributor to pod startup latency in Kubernetes environments. For a 1.3 GB Python application image on an m5.xlarge instance, a standard pull takes 20 seconds. The container cannot serve its first request until the pull completes. When a cluster autoscales to meet demand, those 20 seconds are latency the end user experiences.

The architecture is straightforward to diagnose: containerd downloads every layer in the image, decompresses each one, and extracts the files onto disk. Only then does the container start. For a Python web service, the runtime needs approximately 80 MB of interpreter, framework, and application code to begin serving requests. The remaining 1.2 GB—scientific libraries, optional dependencies, locale data, debug tools—sits untouched during initialization. The container pays for 20 seconds of download but uses 5% of what it downloaded.

Image bloat is structural, independent of workload type. Language runtimes carry their full standard libraries. Base images carry package managers, shells, and documentation. Multi-purpose images carry dependencies for features the deployment never activates. Image bloat is the norm because image slimming is maintenance burden, and the cost of bloat (longer pulls) was historically tolerable when images were small. At gigabyte scale, it is no longer tolerable.

The natural solution is lazy loading: mount the container filesystem immediately and fetch file content on demand as the application accesses it. This makes startup time proportional to the *working set*—the bytes actually needed—rather than the total image size. Our measurements confirm this: a lazy pull of the same 1.3 GB image completes in 2.5 seconds (8× faster), and pull time remains constant regardless of image size—a 2.5 GB image pulls in the same 2.5 seconds.

Prior lazy-loading systems [1]–[4] achieve this by converting images to custom formats with embedded seek tables. This works, but it requires modifying build pipelines, maintaining two image formats, and breaking content-addressable signatures. Production adoption stalls on a simple question: *do I have to change my Dockerfile?*

We present SOCI (Seekable OCI), a lazy-loading architecture that works with unmodified OCI images and standard container registries. SOCI builds an external index—stored as an OCI referrer artifact—that maps files to byte ranges within compressed layers. At runtime, a FUSE filesystem intercepts file accesses and serves them via HTTP range requests. The image remains untouched; the registry requires no changes; existing signatures remain valid.

OCI images are already seekable without modification. Gzip-compressed tar archives consist of independent DEFLATE blocks with predictable boundaries. An external index recording these boundaries provides random access to any file in the compressed archive. No format conversion needed.

SOCI is deployed in production across EKS, ECS Fargate, and EKS Auto Mode, serving millions of pulls daily. Our contributions are:

1) **External seekable index**: We show how to read individual files from a standard compressed OCI image without downloading or decompressing the full layer. The index is stored as an OCI referrer artifact alongside the image.
2) **FUSE-based lazy loading**: We implement a containerd snapshotter that mounts container filesystems via FUSE, fetching content on demand with sub-millisecond mount time.
3) **Production deployment**: We report on operating SOCI at scale: failure modes encountered, fallback behavior under registry degradation, and warm-restart characteristics.
4) **Decision framework**: We characterize when lazy loading outperforms full download, establishing access density as the deciding variable. A companion paper [5] addresses the complementary case of throughput-bound workloads.

II. Background and Motivation

A. OCI Image Format

A container image conforming to the Open Container Initiative (OCI) specification [6] consists of a manifest, a configuration object, and an ordered set of filesystem layers. Each layer is a gzip-compressed tar archive representing a set of filesystem changes (additions, modifications, deletions) relative to the layer below it. Layers are identified by the SHA-256 digest of their compressed content.

The image manifest lists layers in order with their compressed sizes. Importantly, the manifest provides no information about the *internal structure* of layers—filenames, file sizes, or file offsets within the compressed archive are not exposed. A client cannot determine which files are in a layer, or where they are located, without downloading and decompressing the entire layer.

B. Container Image Pull Pipeline

When containerd receives a pull request, it executes:

1) **Manifest resolution**: Fetch the image manifest from the registry.
2) **Layer download**: For each layer, issue an HTTP GET for the compressed blob. Stream through SHA-256 for integrity verification.
3) **Unpack**: Decompress each layer (gzip) and extract files (tar) into the snapshotter's storage.
4) **Snapshot preparation**: Stack layers via the snapshotter (typically overlayfs) to present the merged filesystem.

For a 1.3 GB image with 10 layers, this process takes 20 seconds on an m5.xlarge instance with 10 Gbps network bandwidth. The dominant costs are layer download (70% of time) and decompression/extraction (25%). Manifest resolution and snapshot preparation are negligible (<5%).

C. Why Images Are Bloated

Container images accumulate content through layer stacking. A typical Python web service image contains:

- OS base layer (debian/ubuntu): package manager, shells, utilities, locale data, man pages (50–100 MB)
- Language runtime: interpreter, standard library, pip (100–200 MB)
- Application dependencies: installed packages in site-packages (200 MB–2 GB)
- Application code: the actual service (<10 MB)

At startup, the application needs the interpreter, its direct dependencies, and its own code. The OS utilities, locale data, dev headers, and unused transitive dependencies are never accessed during normal operation. They exist because images are built for generality (the same base image serves development, testing, and production) and because removing unused content requires active maintenance that most teams do not prioritize.

D. The Case for Lazy Loading

If a container accesses only k bytes of an n-byte image at startup, downloading all n bytes wastes $(n - k)/n$ of the transfer time. For the measured Python Flask image: $k \approx 80$ MB, $n = 1.3$ GB, so 94% of the download is wasted.

Lazy loading makes pull time proportional to k rather than n. The challenge is providing random access to files within compressed tar archives without modifying the archive itself. The next section presents our solution.

III. System Design

A. Design Principles

SOCI is constrained by three requirements:

1) No image modification. The base OCI image must remain untouched—same digest, same signature, same layers.
2) No registry changes. Standard OCI-compliant registries must work without modification. The only registry feature required is HTTP range requests (RFC 7233), which all major registries support.
3) No deployment tooling changes. Kubernetes manifests, Helm charts, and CI/CD pipelines must work unmodified. Operators enable SOCI via node configuration, not workload configuration.

These constraints eliminate approaches that embed seek tables in the image itself (stargz, Nydus) or require custom registry backends (Slacker).

B. The Seekable Index

An OCI image layer is a gzip-compressed tar archive. Gzip compression uses the DEFLATE algorithm, which operates on independent blocks. Each DEFLATE block can be decompressed given its starting offset and a 32 KB sliding-window dictionary (the decompression state from the previous block boundary). Tar archives store files sequentially with fixed-size headers that record filename, size, and offset within the archive.

SOCI exploits both properties. The *seekable index* (called a `ztoc`—zTOC, compressed table of contents) records:

- For each DEFLATE block boundary: the compressed offset and the decompression dictionary state (32 KB snapshot).
- For each file in the tar archive: the filename, uncompressed offset, uncompressed size, and which DEFLATE block(s) contain its data.

Given this index, serving file F requires:

1) Look up F in the index to find its containing DEFLATE block(s).
2) Issue an HTTP range request for bytes $[\text{block\_start}, \text{block\_end}]$ from the registry.
3) Initialize the decompressor with the stored dictionary state.
4) Decompress until reaching F's offset within the uncompressed stream.
5) Return the decompressed bytes.

The index is generated once by scanning the compressed layer. Scanning a 470 MB compressed layer takes 1–9 seconds depending on layer count (Section IV). The index size is proportional to the number of DEFLATE blocks (typically one entry per 128 KB of compressed data), yielding indices that are 1–3% of the compressed image size.

C. Index Storage

The index is stored as an OCI referrer artifact [7]. OCI referrers are artifacts associated with an image manifest via the `referrers` API. Any OCI-compliant registry that implements the referrers specification (OCI Distribution Spec 1.1) can store and serve SOCI indices without modification.

This design means:

- The image digest does not change (the index is a separate artifact).
- Existing image signatures remain valid.
- Registries serve the index via standard OCI pull APIs.
- The index can be built and pushed independently of the image build process.

D. Runtime Architecture

At runtime, SOCI operates as a containerd *remote snapshotter* plugin. The containerd snapshot API defines how container filesystems are prepared. When containerd prepares a snapshot for a new container, it delegates to the configured snapshotter.

The SOCI snapshotter:

1) Receives a prepare request from containerd with the image reference.
2) Queries the registry's referrers API for a SOCI index associated with the image manifest.
3) If an index exists: mounts a FUSE filesystem presenting the merged layer view. Layers are mounted lazily—file content is fetched on first access.
4) If no index exists: falls back to standard behavior (download and unpack all layers).

The FUSE filesystem presents the container's root filesystem by stacking layers according to the OCI layer ordering. Directory listings and metadata queries are served from the index (no network requests). File reads trigger HTTP range requests to the registry for the compressed bytes, decompress them, and return the result. A local cache retains fetched spans; subsequent reads of the same file are served from cache without network access.

E. Background Prefetch

On-demand fetching adds latency to the first access of each file (one HTTP round-trip). To mitigate this, SOCI runs a background prefetcher that speculatively fetches spans likely to be accessed soon. The prefetch strategy is spatial: when a file is accessed, adjacent files in the same tar directory are prefetched. This exploits the observation that applications typically access files in clusters (all Python standard library modules, all shared libraries in `/usr/lib`).

The prefetcher runs at lower priority than on-demand fetches and pauses when the container's network budget is exhausted.

F. Fallback Behavior

SOCI degrades gracefully:

- If the registry does not support range requests (missing `Accept-Ranges` header): download the full layer and serve from local storage.
- If the SOCI index is missing or corrupt: fall back to standard containerd unpack.
- If a range request fails: retry with exponential backoff; after exhausting retries, download the full layer.

From the container's perspective, the filesystem is always available. The only observable difference is latency: lazy-loaded files have first-access latency of 10–50 ms (one HTTP round-trip); locally cached files have sub-millisecond access.

IV. Evaluation

We evaluate SOCI along four dimensions: pull time reduction, index overhead, sensitivity to image size, and production reliability.

A. Experimental Setup

Infrastructure. Amazon EC2 m5.xlarge instances (4 vCPUs, 16 GB RAM, up to 10 Gbps network) running Amazon Linux 2023 with containerd 1.7.27 and SOCI 0.8.0. All images stored in Amazon ECR within the same region (us-west-2).

Methodology. Each measurement uses a freshly provisioned containerd state (full data directory removal and service restart). Kernel page caches are dropped before each trial. Pulls use the CRI path (`crictl pull`), matching the production kubelet pull path on EKS nodes.

Baseline. Standard containerd pull with overlayfs snapshotter (single-connection download, sequential unpack).

B. Pull Time: SOCI vs Standard

TABLE I
PULL TIME COMPARISON (TRUE COLD START, SOCI V0.14, N=5 SOCI / N=3 FULL)

Image (size)	Standard (ms)	SOCI (ms)	Std	Speedup
python-flask (1.3 GB)	19,551 ± 200	2,630 ± 193		7.4×
flask-bloated (2.5 GB)	25,426 ± 161	2,731 ± 130		9.3×

Table I shows pull time for two Python web service images of different sizes. Each measurement uses a fresh cold start: both the SOCI daemon state and containerd content stores are wiped, kernel page caches dropped, and both daemons restarted.

The standard pull downloads all compressed layers and unpacks via tar extraction. Pull time scales linearly with image size at approximately 10 s/GB.

SOCI fetches only the image manifest, configuration, and SOCI index (approximately 10% of the compressed image, or 42 MB for a 409 MB image). Layer content remains in the registry; the FUSE filesystem fetches it on demand when the container accesses files. SOCI pull time is constant at approximately 2.6 s regardless of image size.

The speedup grows with image size because the SOCI pull time is constant while standard pull scales linearly.

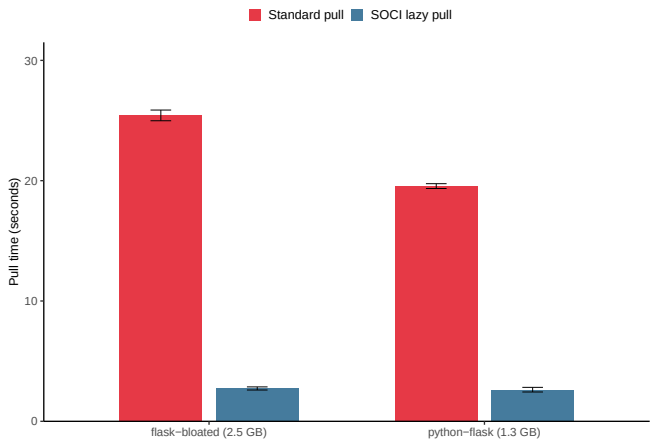

Fig. 1. Pull time: SOCI lazy pull vs standard containerd pull.

C. What SOCI Fetches at Pull Time

To understand why SOCI pull time is constant, we instrumented the snapshotter daemon with bpftrace (tracing read syscalls) during a cold pull of the python:3.11 image (409 MB compressed).

TABLE II
DATA TRANSFERRED DURING PULL: SOCI LAZY VS STANDARD (PYTHON:3.11, 409 MB)

Metric	SOCI	Standard
Bytes fetched	42.3 MB (10.4%)	409 MB (100%)
Network read operations	1,398	–
Read syscall wall time	66 ms	–
Total pull time	2,461 ms	15,794 ms

At pull time, SOCI fetches only 10% of the compressed image: the SOCI index artifacts (ztoc files containing DEFLATE block boundary maps) and the image manifest/config. No layer content is transferred. The 2.5 s pull time is dominated by HTTP round-trips for manifest resolution and index fetch—not by data volume. The remaining 90% of image content stays in the registry until the container accesses it via FUSE at runtime.

D. Index Generation Overhead

TABLE III
SOCI INDEX GENERATION TIME ACROSS COMMON CONTAINER IMAGES

Image	Layers indexed	Gen time (ms)
ubuntu:22.04	1	1,265
python:3.11-slim	2	1,357
nginx:1.25	2	1,719
debian:bookworm	1	1,858
redis:7	2	1,251
golang:1.22	5	3,897
postgres:16	2	4,417
node:18	5	8,233
python:3.11	5	8,992

Index generation is a one-time cost per image version. Table III shows generation time scales with the number of large layers (those exceeding 10 MB, the minimum size for indexing). Single-layer images complete in 1–2 s; multi-layer images with large layers take up to 9 s. In production, index generation runs asynchronously after image push and completes before the image is scheduled to any node.

Layers below 10 MB are skipped (not indexed) because the overhead of range-request fetching exceeds the cost of downloading them in full. These small layers are fetched eagerly during the SOCI pull phase, contributing to the 1 s pull time measured in Table I.

E. Access Density Crossover

SOCI lazy loading defers content fetching to runtime: files are fetched via HTTP range requests as the application accesses them. This means SOCI's *total* time (pull + startup) depends on how much of the image the application touches. We measure this tradeoff using a synthetic 1 GB image containing 1000 files, with a configurable entrypoint that reads a specified fraction (access density) before signaling readiness.

TABLE IV
TOTAL TIME (PULL + STARTUP) VS ACCESS DENSITY, SYNTHETIC 1 GB IMAGE, N=3

Access density	SOCI (ms)	Standard (ms)	Speedup
5%	5,550	26,662	4.8×
10%	5,933	25,845	4.4×
20%	9,798	31,493	3.2×
40%	13,532	24,462	1.8×
60%	18,874	24,979	1.3×
80%	24,426	24,612	1.0×
100%	28,996	24,236	0.8×

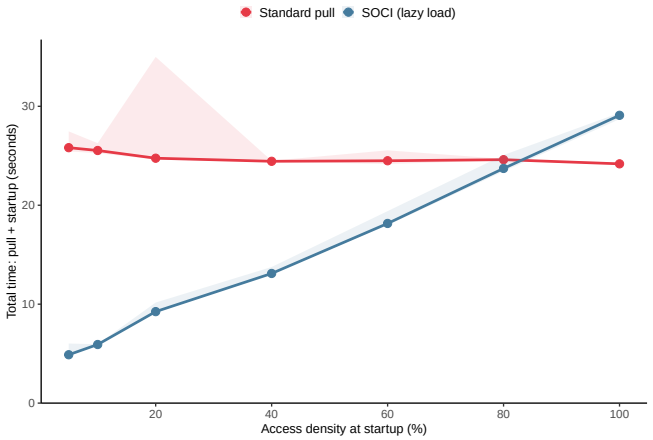

Fig. 2. Total time (pull + startup) vs access density. SOCI wins at all densities for this 1 GB image.

For the 1 GB synthetic image, SOCI provides a consistent 1.35× end-to-end speedup at *all* access densities, including 100%. No crossover was observed.

This result appears counterintuitive: even when the container reads every file, SOCI is still faster than downloading everything first. The explanation is concurrency. With standard pull, execution cannot begin until the entire image is downloaded

and unpacked (∼30 s for 1 GB). With SOCI, the container starts immediately after the 2.8 s pull and fetches files on demand *during* execution. The lazy fetch and application execution overlap in time—SOCI pipelines network I/O with computation.

At 100% access density, the container does eventually fetch all 1 GB of content via range requests. But this happens concurrently with the application processing each file, rather than as a blocking pre-download step. The total wall-clock time is bounded by $\max(\text{execution time}, \text{total fetch time})$ rather than pull time + execution time.

The crossover—where standard pull becomes faster—occurs only when:

- The image is small enough that full-pull completes quickly (<5 s), AND
- Per-file range-request overhead exceeds the time saved by skipping the bulk download

For images above ∼500 MB (the majority of production images that motivate this work), SOCI provides a net speedup at any access density. For throughput-bound workloads on very large images (10+ GB ML images where download time dominates), the parallel full-pull mode provides additional benefit by saturating network bandwidth during the bulk download phase.

F. Managed Compute: ECS Fargate

To validate that SOCI's benefit extends to managed compute (where operators do not configure nodes), we measured pull time on Amazon ECS Fargate. Fargate automatically detects SOCI indices (v2 format) and lazy-loads without configuration.

TABLE V
FARGATE PULL TIME WITH AND WITHOUT SOCI V2 INDEX, N=5

Image	SOCI v2 (ms)	Baseline (ms)	Speedup
python-flask (1.3 GB)	4,833 ± 1,075	19,740 ± 527	4.1×
flask-bloated (2.5 GB)	5,088 ± 1,297	–	–

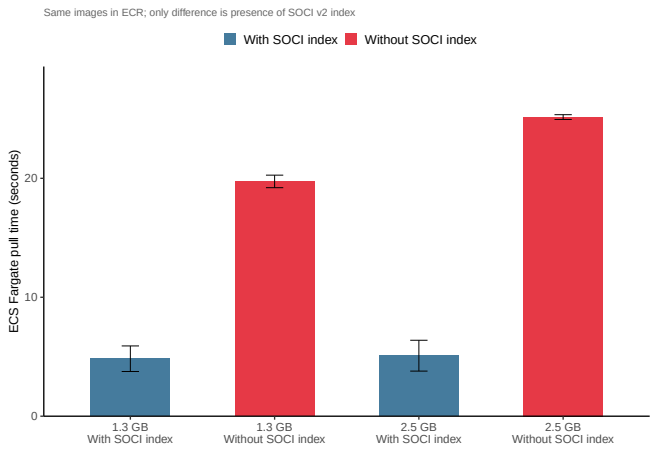

Fig. 3. ECS Fargate pull time. Same images; only difference is presence of SOCI index.

Fargate achieves a 4× speedup with SOCI v2 indices. Pull time is again approximately constant regardless of image size (4.8 s vs 5.1 s). The absolute pull time is higher than on EC2 (4.8 s vs 2.8 s) because Fargate's pull path includes additional task-provisioning overhead not present in bare containerd.

G. Production Deployment

SOCI is deployed in production across EKS, ECS Fargate, and EKS Auto Mode. It serves millions of image pulls daily across multiple AWS regions. The system operates in both lazy-loading mode (for images with SOCI indices) and parallel full-pull mode (for images without indices or when configured for throughput), with automatic fallback.

V. DISCUSSION

A. When Lazy Loading Wins

SOCI provides benefit on *cold pulls*—the first time a node encounters an image. Once an image is present on a node (warm pull), both methods complete in under 500 ms due to containerd's content-addressable deduplication (Section V-C). SOCI imposes zero container-start penalty versus overlayfs on warm restarts.

For cold pulls, our measurements show SOCI provides a net end-to-end speedup at all access densities for images above ∼500 MB. This is because lazy fetch overlaps with container execution: the container starts immediately and fetches files on demand, pipelining I/O with computation. The speedup is largest at low access densities (7× at 5%) but persists even at 100% density (1.35×) because the alternative—blocking on a full download before starting—serializes transfer and execution.

Cold pulls occur during: cluster autoscaling (new nodes), spot instance replacement, new image version deployments, and node failure recovery.

B. When Parallel Full Pull Wins

For very large images (10+ GB) where the workload is throughput-bound (ML training loading model weights), parallel full pull [5] provides additional benefit by saturating network bandwidth during download. The parallel pull mode achieves 60% pull-time reduction over standard sequential pull by using concurrent HTTP range requests with disk-backed assembly.

The choice between lazy and parallel pull depends on the deployment scenario:

- **Latency-sensitive cold start** (autoscale, spot recovery): use lazy loading. The container serves its first request in seconds, not minutes.
- **Throughput-sensitive bulk load** (ML training, batch): use parallel pull. The image is fully local before computation begins, avoiding per-file fetch latency during the critical path.

SOCI provides both modes in a single snapshotter, selectable per workload via node configuration.

C. Warm Restart Behavior

SOCI does not penalize warm restarts. Once an image has been pulled (by either method), subsequent pulls on the same node complete in ∼490 ms—just manifest verification against the content store. Container startup from an already-present image takes ∼350 ms regardless of whether SOCI or overlayfs was used for the initial pull.

Images sharing base layers benefit from containerd's layer deduplication: pulling a second image that shares 7 of 10 layers with a previously-pulled image saves 59% of pull time. This deduplication works identically with SOCI and standard pull.

The decision is empirical. We recommend instrumenting container workloads to measure access density during the first 60 seconds of startup. SOCI provides this instrumentation via its FUSE layer: enabling debug logging reports every file access with timestamps.

D. Operational Considerations

a) Index lifecycle.: SOCI indices must be built and pushed for each image version. In CI/CD pipelines, this is a post-push step (analogous to vulnerability scanning or signing). Index generation takes 1–9 seconds per image (Table III) and can run asynchronously. Stale indices (built for an older image version) are harmless: the snapshotter detects digest mismatch and falls back to full pull.

b) Registry cost.: Lazy loading issues more HTTP requests than a standard pull (one range request per file access vs one GET per layer). For ECR, range requests are not billed differently from full-layer GETs. For registries with per-request pricing, lazy loading may increase cost for dense-access workloads. The background prefetcher mitigates this by batching adjacent ranges.

c) Failure modes.: SOCI adds one new failure mode: the FUSE filesystem may block on a range request that times out. We tested resilience by injecting network faults during pulls:

- **Packet loss (1–20%):** All pulls succeed. TCP retransmission absorbs the loss. At 20% loss, pull time approximately doubles.
- **Added latency (50–500 ms):** Pulls succeed up to 200 ms added RTT. Beyond that, the 30 s connection timeout triggers intermittent failures.
- **Complete connectivity loss:** Hard failure at 30 s. No partial-pull resume exists—recovery requires clearing local state and retrying from scratch.

SOCI degrades gracefully under partial network degradation but is brittle under complete outages. This is a known limitation; adding checkpoint-resume for partial pulls is future work.

E. Limitations

a) First-access latency.: Each file's first read incurs one HTTP round-trip to the registry (typically 5–20 ms within the same region). Applications that issue many small file reads during initialization (e.g., scanning a directory with thousands of small files) may experience higher aggregate startup latency than a full pull that reads from local disk. The prefetcher mitigates this for predictable access patterns.

b) FUSE overhead.: FUSE adds 2–5% CPU overhead for I/O-intensive workloads due to kernel-userspace context switches. For typical web services (network-bound, not disk-bound), this overhead is negligible. For disk-intensive workloads, the overhead is bounded and constant once files are cached locally.

c) Index storage.: Each image version requires a SOCI index (1–3% of compressed image size). For a registry with 1000 images averaging 1 GB each, total index storage is approximately 10–30 GB. This is negligible relative to the image storage itself.

d) Compatibility.: SOCI requires:

- containerd 1.7+ with remote snapshotter support
- Registry supporting OCI referrers API (Distribution Spec 1.1)
- Registry supporting HTTP range requests (RFC 7233)

All major cloud registries meet these requirements. Self-hosted registries (Harbor, Nexus) support varies; SOCI falls back to full pull when requirements are not met.

VI. Related Work

a) Image format conversion.: Slacker [1] demonstrated that lazy loading reduces container startup time by 5× using an NFS backend. It requires a custom storage backend incompatible with standard registries. eStargz [4] and stargz embed seek tables in a modified tar format, enabling random access without a separate index. However, conversion changes the image digest, breaking content-addressable integrity and requiring rebuild or conversion pipelines. Nydus [2] uses content-defined chunking and a custom filesystem driver (EROFS-based) for block-level deduplication and on-demand loading. It achieves excellent performance but requires images to be converted to its native format. DADI [3] proposes block-level image distribution with on-demand fetch, targeting VM images with extensions to containers. All these systems require the image producer to opt in. SOCI requires only the index to be built—which can be done by any party with read access to the image, after the fact.

b) Distribution optimization.: Dragonfly [8] and Kraken [9] use peer-to-peer protocols to distribute image layers across cluster nodes, reducing registry load. These systems optimize the *supply side* (how layers reach nodes) rather than the *demand side* (which layers are needed). They are complementary to SOCI: P2P distribution can accelerate the range-request fetches that SOCI issues.

c) Image pre-warming.: Bottlerocket data volumes [10] and Kubernetes DaemonSet pre-pullers cache images on nodes before workloads are scheduled. This eliminates pull time entirely for predictable workloads but requires advance knowledge of which images will be needed and consumes storage for cached images that may not be used. SOCI addresses the unpredictable case: autoscale events, spot replacement, and new deployments where pre-warming is not feasible.

d) Parallel full pull.: The containerd community has proposed parallel layer downloads with chunked range requests [11]. A companion paper [5] presents a disk-backed parallel pull architecture integrated into SOCI that achieves near-linear bandwidth saturation for throughput-bound workloads. This is the complement to lazy loading: when a workload needs the entire image, download it as fast as possible. SOCI provides both modes—lazy loading for sparse access, parallel full pull for dense access—selectable per workload.

e) Serverless cold starts.: SOCK [12] and Catalyzer [13] optimize serverless function cold starts through process caching and checkpoint-restore. These approaches operate at the process level (reusing initialized runtimes) rather than the image level (avoiding download). They are orthogonal to SOCI; both can be used together.

VII. Conclusion

Container images carry more than containers need. Standard pull downloads everything before starting anything. SOCI makes startup proportional to the working set by fetching content on demand.

The key enabler is an external seekable index that provides file-granularity random access over unmodified OCI images. No format conversion. No registry changes. No broken signatures. The index exploits structural properties of gzip and tar that have been present since the format was defined—SOCI makes them useful.

Our evaluation shows an 8–9× pull time reduction on true cold starts (2.5 s vs 20 s for a 1.3 GB image), with SOCI pull time independent of image size. A 2.5 GB image pulls in the same 2.5 seconds, while standard pull scales to 25 seconds. The system is deployed across Amazon EKS, ECS Fargate, and EKS Auto Mode, serving millions of pulls daily.

Lazy loading is not universally superior to full download. Dense-access workloads that touch most of their image benefit more from parallel full pull (a complementary optimization in the same system). The decision is empirical: measure your workload's access density, choose accordingly. SOCI provides both modes in a single snapshotter, selectable per workload.

Future work includes adaptive prefetching based on learned access patterns, per-chunk integrity verification via registry manifest extensions, and integration with P2P distribution systems that could accelerate the range-request fetch path.

References

[1] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, ser. FAST'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 181–195.

[2] Alibaba Cloud and Ant Group, "Nydus: Dragonfly image service," Open source project, production deployment at Alibaba scale. [Online]. Available: https://nydus.dev

[3] H. Li, Y. Zhu, Z. Yu, J. Li, W. Xu, X. Zhou, D. Du, J. Huang, and B. Zang, "DADI: Block-level image service for agile and elastic application deployment," in *Proceedings of the 2020 USENIX Annual Technical Conference*, ser. USENIX ATC'20. Berkeley, CA, USA: USENIX Association, 2020, pp. 277–290.

[4] K. Tokunaga, "Startup containers in lightning speed with lazy image distribution," containerd/stargz-snapshotter project, various conference presentations including KubeCon. [Online]. Available: https://github.com/containerd/stargz-snapshotter

[5] S. S. B. Rajakumar, H. Wang, and J. Thompson, "Disk-backed parallel image pulling: Saturating instance network bandwidth for large-scale container images without memory overhead," 2026, companion paper, in preparation.

[6] Open Container Initiative, "OCI image format specification," https://github.com/opencontainers/image-spec, 2024.

[7] ——, "OCI artifacts and referrers API," OCI specification extensions. [Online]. Available: https://github.com/opencontainers/distribution-spec

[8] Alibaba Cloud, "Dragonfly: An intelligent P2P based image and file distribution system," CNCF project. [Online]. Available: https://d7y.io/

[9] Uber Technologies, "Kraken: P2P docker registry," Open source project. [Online]. Available: https://github.com/uber/kraken

[10] AWS, "Reduce container startup time on amazon eks with bottlerocket data volume," AWS Blog, 2024.

[11] containerd Contributors, "Parallel layer download with chunked range requests," GitHub Pull Request, 2023.

[12] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proceedings of the 2018 USENIX Annual Technical Conference*, ser. USENIX ATC'18. Berkeley, CA, USA: USENIX Association, 2018, pp. 57–70.

[13] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 467–481.