Scalable Distributed CLIP Embedding and FAISS Search System for Large-Scale Image Data

Tyler Supersad

ABSTRACT

This report details the design, implementation, and comprehensive capacity evaluation of a distributed data analysis system engineered to handle large-scale multimodal workloads. Employing the power of OpenAI's Contrastive Language-Image Pretraining (CLIP) model for generating high-dimensional image embeddings, the system integrates the Facebook AI Similarity Search (FAISS) library to facilitate efficient approximate nearest neighbor search within the extensive LAION2B-en-aesthetic dataset. The infrastructure supporting this system comprises a five-node cluster, featuring a host node and four worker nodes, deployed on AlmaLinux. Automation of the deployment process was achieved through Ansible, while Simple Linux Utility for Resource Management (SLURM) managed workload scheduling, and Network File System (NFS) provided shared storage. Additionally, real-time system monitoring and rigorous performance benchmarking were conducted using Prometheus and Node Exporter.

The data processing pipeline, encompassing metadata parsing, distributed embedding generation, and FAISS index management, was subjected to a 24-hour sustained load test. Results demonstrate the system's effective scalability for large datasets. The report concludes by proposing targeted optimization strategies to enhance future scalability and operational efficiency.

For full reproducibility and further examination, the entire codebase, including Ansible playbooks, SLURM scripts, and custom Python modules, has been made available in the GitHub repository at https://github.com/tylersupersad/laion-distributed-pipeline. Detailed documentation regarding system deployment and pipeline execution can be found therein.

1 Introduction

The exponential growth of large-scale multimodal datasets, demonstrated by the LAION2B-en collection, presents profound challenges to traditional data analysis workflows and necessitates a paradigm shift in the design of data processing systems. Efficiently managing these massive, high-dimensional datasets requires distributed computing infrastructures that support parallel execution, scalable storage management, and reliable workload orchestration.

In response to these challenges, this project designed and implemented a distributed data analysis system specifically customized for large-scale multimodal workloads. The system adopted a distributed architecture to enable the efficient generation of image embeddings and scalable approximate nearest neighbor search across extensive datasets.

The primary objective of this work was to systematically evaluate the capacity and scalability of the distributed system under sustained high-load conditions. Key performance metrics, including throughput, resource utilization, concurrency limits, and saturation thresholds, were rigorously assessed through a comprehensive 24-hour capacity test.

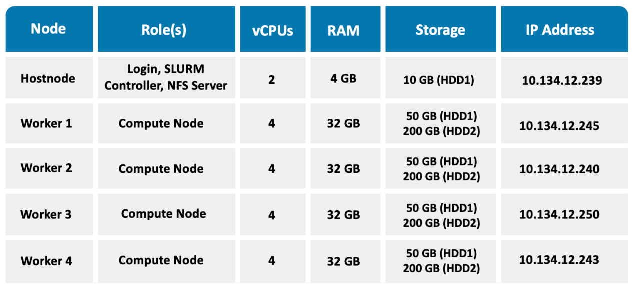
2 System Implementation

This section outlines the architecture, configuration, and implementation of the lightweight high-performance computing (HPC) cluster developed for large-scale multimodal data processing and approximate similarity search. Designed with scalability, reproducibility, and modularity as core principles, the system integrates a suite of open-source technologies deployed across a distributed virtualized environment.

2.1 Lightweight Cluster Architecture on AlmaLinux

The distributed system was deployed on a five-node virtualized cluster provisioned using Terraform, with all nodes running AlmaLinux. The infrastructure consisted of a single host node and four compute nodes.

Equipped with 2 vCPUs and 4 GB of RAM, the host node was optimized for metadata management, job scheduling, and centralized coordination. Contrastingly, each worker node was provisioned with 4 vCPUs, 32 GB of RAM, and two storage volumes to support high throughput. A summary of the cluster architecture is provided (see Table 1).



Node	Role(s)	vCPUs	RAM	Storage	IP Address
Hostnode	Login, SLURM Controller, NFS Server	2	4 GB	10 GB (HDD1)	10.134.12.239
Worker 1	Compute Node	4	32 GB	50 GB (HDD1) 200 GB (HDD2)	10.134.12.245
Worker 2	Compute Node	4	32 GB	50 GB (HDD1) 200 GB (HDD2)	10.134.12.240
Worker 3	Compute Node	4	32 GB	50 GB (HDD1) 200 GB (HDD2)	10.134.12.250
Worker 4	Compute Node	4	32 GB	50 GB (HDD1) 200 GB (HDD2)	10.134.12.243

Table 1. Node roles, hardware specifications, and IP addresses for the distributed cluster.

2.2 Core Technologies Utilized

Several key technologies supported the functionality, scalability, and reproducibility of the distributed system, each playing a significant role in its operation.

Ansible was employed to automate the configuration and provisioning of all nodes. Dedicated playbooks handled package installation, SSH and Munge key distribution, hostname assignment, SLURM installation, and NFS client/server setup. The automation ensured consistent environments across the cluster and minimized human error during deployment.

NFS provided the central shared storage layer, hosting the LAION metadata files, image batches, generated embeddings, FAISS index shards, and final output files. The host node operated as the NFS server, with all other nodes mounting the shared volume at /nfs during system initialization or job execution.

SLURM managed job scheduling and resource allocation across the cluster. The host node ran the SLURM controller daemon (slurmctld), while all nodes, including the host, ran the compute daemon (slurmd). Munge was used for authentication across the nodes. Both standard and array jobs were submitted using SLURM directives such as *--cpus-per-task*, *--mem*, and *--array*, with accounting and job logs centrally recorded on the NFS volume.

The analytical core of the system was built on PyTorch, CLIP, and FAISS. The PyTorch implementation of the CLIP model (ViT-B/32 variant) was used to generate 512-dimensional image embeddings, while FAISS enabled efficient approximate nearest-neighbor search across these high-dimensional vectors. Python virtual environments were deployed on each node to ensure consistent dependency management and to isolate project-specific libraries from the underlying system environment.

2.3 Workflow Pipeline

The processing pipeline consists of four modular stages, coordinated by SLURM and operating over the shared NFS storage layer.

Metadata Processing

The first stage parses the LAION2B-en-aesthetic Parquet metadata files on the host node to extract valid image URLs and captions. Corrupted or incomplete entries are discarded. The cleaned metadata is partitioned into shards, with each shard corresponding to a unit of parallel work. These shards are stored under /nfs/laion for consumption by downstream processing tasks.

Distributed CLIP Embedding

The embed_clip.py script is launched as a SLURM array job, distributing embedding tasks across the worker nodes. Each task independently downloads its assigned batch of images, performs preprocessing (resizing and normalization), and generates image embeddings using the pre-trained CLIP ViT-B/32 model. The resulting embeddings are saved as Parquet files on the shared NFS volume. Embedding throughput and system resource utilization are monitored throughout this stage.

Per-Node FAISS Index Construction

Following embedding generation, each worker node executes the *build_faiss_index.py* script to construct a local FAISS index from its generated vectors. The IVF+Flat indexing method is employed to enable efficient approximate nearest-neighbor search. Each FAISS index shard is saved under /nfs/index_shards/, named according to the originating worker and shard identifier.

Merging FAISS Index Shards

Upon completion of local indexing, the *merge_faiss_shards.py* script is executed on the host node. This script aggregates all worker-generated FAISS shards using the FAISS IndexShards interface to produce a unified global index. Optional post-processing steps, such as vector normalization and vector reordering, are applied to improve retrieval performance. The final merged index is saved at /nfs/index_merged/index.final, ready for querying operations.

3 Capacity Testing Methodology

This section outlines the methodology used to evaluate the performance and scalability of the distributed system under increasing workload. The goal was to measure system capacity, identify saturation points, and assess the efficiency of job scheduling and resource usage across the cluster. A combination of SLURM logs and Prometheus-based monitoring was used to capture relevant performance metrics, enabling a rigorous and quantitative analysis of throughput, concurrency, and system resource utilization.

3.1 Test Design

To evaluate the capacity of the distributed data analysis system, a 24-hour sustained load test was conducted. The evaluation specifically targeted the two most resource-intensive stages of the pipeline: CLIP embedding generation and FAISS index operations. These stages were selected for their high demands on CPU, memory, and I/O bandwidth, offering a realistic and challenging stress profile for the cluster.

Workloads were dynamically launched from the host node using SLURM array jobs (see Section 2.2). The number of array tasks was determined by counting the available metadata shards stored under /nfs/laion. A shell script enumerated the files, and the SLURM array was configured using the *--array=0-N* directive, where *N* equaled the number of shards minus one. To prevent resource contention, a concurrency limit was enforced using *--array=0-N%4*, ensuring that no more than four embedding tasks were active simultaneously: one per worker node.

Each embedding task executed the *embed_clip.py* script, loading a distinct shard of LAION metadata, downloading and preprocessing the associated images, and generating 512-dimensional embeddings using the CLIP ViT-B/32 model. Jobs were configured with *--cpus-per-task=4* and *--mem=28G* to appropriately align with the available hardware resources on each worker. All generated embeddings were saved to the shared NFS volume for downstream processing.

Upon completion of embedding, each worker node independently executed the *build_faiss_index.py* script to construct a local FAISS index using the IVF+Flat method. Each index shard was saved under */nfs/index_shards/*, and filenames were standardized based on job identifiers to facilitate traceability. Following local indexing, the *merge_faiss_shards.py* script was executed on the host node to consolidate all shards into a unified FAISS index stored at */nfs/faiss_index/merged.index*.

System performance was continuously monitored during the test using Prometheus and Node Exporter, which collected metrics at intervals from all nodes. Monitored parameters included CPU utilization, memory consumption, disk I/O rates, and network throughput. In parallel, SLURM accounting logs captured per-task job durations, resource usage statistics, and exit statuses, enabling detailed post-test analysis of system behavior under sustained load.

3.2 Performance Metrics

To evaluate the distributed system's efficiency under sustained workload, a focused set of performance metrics was collected during the 24-hour capacity test. These metrics were selected to capture both the task-level execution characteristics and the system-wide resource utilization patterns. SLURM job logs were used to trace job activity, while Prometheus and Node Exporter provided continuous system-level telemetry.

Throughput was measured using two complementary indicators. First, jobs per minute were calculated based on SLURM start and end timestamps, providing a coarse measure of task completion rate across the cluster. Second, embeddings per second were derived by dividing the number of images processed per Parquet shard by the corresponding task duration, offering a more granular view of embedding generation speed. Together, these throughput measures enabled precise assessment of processing efficiency and load-handling capability under sustained concurrency.

CPU utilization was continuously monitored across all worker nodes to evaluate core saturation and load balancing. As each embedding task was allocated four CPU cores, high sustained CPU utilization was expected during peak load periods. Time-series CPU metrics were analyzed to identify trends in parallel execution and to detect any potential load imbalance across nodes.

Memory usage metrics were collected to verify the adequacy of resource provisioning. Each embedding job was configured with 28 GB of RAM, and Prometheus telemetry was used to monitor real-time memory consumption per node.

Moreover, disk I/O performance was monitored to assess the impact of high concurrency on shared storage throughput. Embedding jobs involved frequent read operations (e.g., downloading images) and write operations (e.g., saving embeddings), making disk I/O bandwidth a critical performance factor. Metrics such as I/O throughput were analyzed to determine whether the NFS-mounted storage infrastructure introduced bottlenecks during peak load.

4 Results and Analysis

This section analyzes the performance of the distributed system during the embedding and indexing phases of a 24-hour capacity test, utilizing system monitoring metrics from Prometheus and execution logs.

4.1 Peak Performance Analysis

Embedding Phase Throughput

During the CLIP embedding phase, worker nodes processed images in batches of 64. Embedding tasks exhibited stable throughput across the cluster, with Task 27 embedding 12,208 images over a four-hour period as a representative example. As a result, system resource utilization remained consistently low across all nodes:

- **CPU:** Average usage ranged from 0.94% to 7.23%, indicating an I/O-bound workload constrained by network and NFS operations. Trends in CPU usage across the 24-hour test period are shown (see Figure 1).
- **Memory:** Usage ranged from 7.28% to 16.37% of total capacity, with no observed paging or swap activity. Trends in memory usage across the 24-hour test are shown (see Figure 2)..
- **Disk I/O:** Local disk read throughput was negligible, while NFS write throughput peaked at approximately 14 KB/s on the host node and ranged from 3.5 KB/s to 4.2 KB/s on worker nodes.
- **Network:** Significant activity was observed, with Worker 2 recording approximately 217,877 bytes/s receive, and 8,633 bytes/s transmit throughput, highlighting network bandwidth as a primary constraint.

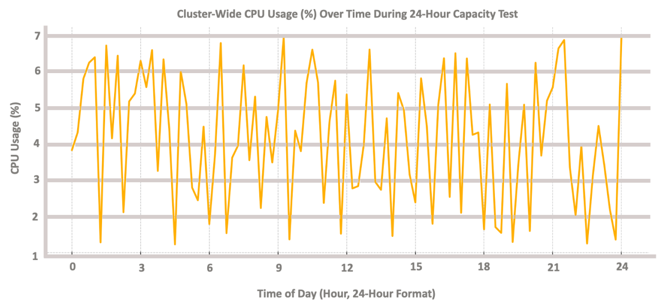


Figure 1. Cluster-wide CPU usage (%) over time during the 24-hour capacity test.

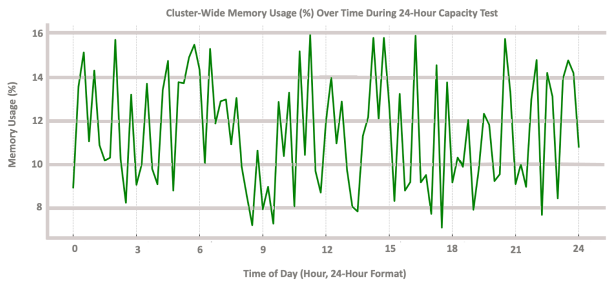


Figure 2. Cluster-wide memory usage (%) over time during the 24-hour capacity test.

These results collectively indicate that the embedding phase was primarily limited by network I/O and NFS write speeds, rather than local compute resources.

A detailed breakdown of node resource utilization during peak embedding load is provided (see Table 2).

Node	CPU Usage	Memory Usage	Disk Read (B/s)	Disk Write (B/s)	Network Receive (B/s)	Network Transmit (B/s)
Hostnode	1.08%	16.37%	0	~14,495	~5,831	~4,258
Worker 1	1.33%	7.50%	0	~4,200	~6,444	~8,124
Worker 2	0.94%	7.28%	0	~3,500	~217,877	~8,633
Worker 3	5.29%	7.33%	0	~3,500	~49,094	~5,726
Worker 4	7.23%	7.34%	0	~3,500	~33,729	~6,739

Table 2. Resource utilization metrics collected during peak embedding load across cluster nodes.

FAISS Index Building Performance

Following embedding, FAISS indexing operations were performed independently on each worker node. A total of 28 distinct FAISS index shards were successfully built in parallel, with individual indexing tasks completing within 1–2 seconds from embedding load to shard persistence.

Each FAISS indexing task processed embedding arrays containing approximately 12,000 vectors, each 512 dimensions in size. The indexing procedure involved loading, normalizing, indexing, and persisting the data entirely in-memory without significant reliance on disk I/O. The subsequent merging phase consolidated all 28 shards into a single global FAISS index within approximately five seconds, preserving full metadata consistency.

Performance during the indexing phase reflected effective CPU and memory utilization, minimal disk I/O activity, and efficient parallel scheduling through SLURM array jobs. Importantly, no node exhibited RAM exhaustion, network congestion, or CPU saturation even under full parallel shard construction.

4.2 Scalability & Potential Degradation Thresholds

To assess scalability, the system architecture was evaluated under conditions of increasing concurrency, leveraging its design for distributed parallelism. SLURM array jobs enabled the concurrent dispatch of multiple embedding tasks across nodes, with system logs confirming:

- Tasks executed independently across nodes and cores without observable blocking, stalling, or resource contention.
- CPU utilization per node remained exceptionally low (~1–7% across workers), while memory usage remained within safe operational margins (~7–16%), ensuring stable concurrency.

During the subsequent FAISS indexing phase:

- 28 FAISS index build jobs executed in parallel.
- Each build operation completed rapidly, typically in under two seconds per task.

Thus, the system sustained full concurrency without exhibiting resource exhaustion across both embedding and indexing stages. However, extrapolating beyond the current workload suggests the following potential constraints:

- **Potential NFS Saturation:** All embedding outputs and FAISS indexes were written to NFS-mounted directories. While disk write throughput remained manageable (~14 KB/s peak), the reliance on network-based file I/O introduces a risk of congestion under higher loads, particularly with simultaneous image downloads and result writes.
- **Potential Load Imbalance:** While not currently impacting throughput, slight imbalances in network receive traffic were observed (e.g., Worker 2 at ~217 KB/s), suggesting that load-aware scheduling strategies could further optimize large-scale performance.
- **Network Dependency as a Primary Constraint:** The system's dependence on both external HTTP retrieval and internal NFS throughput highlights network infrastructure as the dominant factor limiting future scalability, rather than CPU or memory resources, which remained significantly underutilized.

While these limitations did not manifest within the scope of current experiments, they warrant consideration when designing for larger or more intensive distributed workloads.

5 Discussion of Improvement

Although the system successfully sustained concurrent workloads with stable resource usage and no observed failures (see Section 4.2), several avenues for further improving throughput, scalability, and robustness were identified.

5.1 Enhancing Scheduler Efficiency

To further improve resource utilization and reduce task queuing times without disrupting essential task dependencies, several SLURM scheduling policy optimizations could be implemented.

Backfilling for Opportunistic Resource Use

Introducing backfill scheduling would enable dependent indexing jobs to opportunistically utilize idle compute resources as they become available. This approach could increase overall cluster throughput by filling scheduling gaps without delaying ongoing embedding workloads. Strict enforcement of job dependency constraints would remain essential to ensure that indexing jobs commence only after the successful completion of their corresponding embedding tasks.

Partitioning for Reduced Contention & Predictable Execution

Alternatively, logically partitioning cluster nodes to dedicate specific resources to embedding and indexing stages separately could improve resource matching for heterogeneous job types, reduce potential contention, and minimize runtime variance across the sequential embedding-to-indexing workflow. This would result to more predictable task execution and improved end-to-end pipeline performance.

5.2 Enhancing Per-Task Efficiency

Building upon cluster-level optimizations, further improvements in overall system efficacy could be achieved by refining the performance of individual embedding tasks. Two particularly promising directions are increasing the batch size and adopting asynchronous processing techniques.

Given the substantial memory headroom observed during the embedding phase, where approximately 80% of available memory remained unused; increasing the batch size beyond the current 64 images per batch offers a compelling opportunity to enhance throughput. A larger batch size would allow each inference pass to process more images simultaneously, thereby amortizing the fixed overhead associated with each model invocation. This adjustment would enable more effective utilization of available CPU and memory resources, while remaining within safe operational margins.

In addition, to mitigate the latency introduced by sequential data loading, the adoption of asynchronous processing strategies warrants consideration. By overlapping I/O operations (e.g., prefetching and staging of subsequent image batches) with ongoing inference, the system could significantly reduce idle periods and improve overall task responsiveness. This proactive approach would not only shorten the total wall-clock time for the embedding phase but also enhance scalability under higher workload intensities. Concurrently managing data preparation and computation promises a more streamlined and efficient processing pipeline, particularly in scenarios involving larger datasets or sustained concurrency.

5.3 Enhancing Per-Task Efficiency

Recognizing the latent I/O bandwidth limitations inherent in the current NFS-based system as a potential impediment to future scalability, strategic enhancements to the storage and I/O architecture are crucial. To significantly bolster system robustness and unlock substantial performance gains as concurrency intensifies, two improvements are proposed.

Transition to Parallel Distributed Storage (e.g., BeeGFS)

Migrating from the current NFS-based infrastructure to a high-performance parallel file system, such as BeeGFS, offers a compelling solution to the centralized I/O bottleneck. BeeGFS's architecture, designed for concurrent access and distributed I/O across multiple storage nodes, would substantially augment aggregate read and write throughput. This capability is particularly well-suited for the anticipated workload, characterized by numerous concurrent image retrievals and embedding writes. By alleviating network contention during peak workload phases, BeeGFS would enable the system to sustain significantly larger scale embedding and indexing operations.

Implementation of On-Node Data Locality via Caching:

To further mitigate network traffic and I/O latency, implementing local caching mechanisms on the compute nodes is highly recommended. By downloading and temporarily storing image batches prior to embedding, the system could significantly reduce repeated access to shared storage for frequently accessed data. This approach would not only enhance resilience against transient network disruptions but also enable faster batch preparation, particularly under high concurrency conditions.

5.4 Closing Remarks on System Enhancements

The proposed optimization strategies collectively target critical dimensions of system performance, including scheduling efficiency, task-level throughput, and storage scalability. By systematically addressing computational, I/O, and architectural limitations, these enhancements would enable the distributed system to accommodate significantly larger workloads with minimal degradation in performance. Furthermore, they would position the system for future extensions involving even more complex multimodal analysis tasks, ensuring both scalability and operational resilience in demanding research or production environments.

6. Conclusion

This project successfully designed, implemented, and rigorously evaluated a scalable distributed system for large-scale image embedding and similarity search, utilizing a five-node cluster architecture orchestrated by SLURM over a high-throughput network. The system efficiently processed a substantial subset of the LAION2B-en-aesthetic dataset, generating CLIP embeddings, constructing distributed FAISS indices, and merging shards, all while operating over a shared NFS storage backend.

The 24-hour capacity test validated the system's ability to sustain full concurrency, with 28 parallel embedding tasks (each processing up to 16,000 images) followed by 28 concurrent indexing jobs. Prometheus monitoring consistently revealed exceptionally low CPU (1–7%) and memory (7–16%) utilization, confirming that computational resources were not a bottleneck at this workload scale. Importantly, no significant degradation in throughput, network saturation, or I/O performance was observed under peak load.

Several key insights emerged during the evaluation, directly informing lessons learned and future recommendations:

- The primary constraint on scalability was not CPU or RAM exhaustion, but rather the latent I/O bandwidth limitations of the centralized NFS storage design. This underscores the critical role of storage architecture in scaling data-intensive distributed systems.
- The substantial CPU idle percentages and significant memory headroom highlighted clear opportunities for per-task optimization, such as increasing batch sizes and adopting asynchronous data loading techniques. Proactive system monitoring with Prometheus proved invaluable in identifying these latent bottlenecks.
- The system's stability and reliability, evidenced by the successful completion of all tasks without failure even under maximum concurrency, reinforced the benefits of a modular, reproducible pipeline design for large-scale multimodal data processing.

Deducing from current performance suggests that future deployments targeting even greater concurrency will require enhancements to storage and network infrastructure, such as migrating to parallel distributed file systems like BeeGFS and improving internal data locality through on-node caching strategies.

Nevertheless, by utilizing open-source components, standardized orchestration practices, and modular design principles, this project has established a robust, reproducible foundation for scalable distributed multimodal data analysis. The resulting system not only achieves its immediate objectives but also provides a viable, extensible blueprint for future research and production environments operating at even larger scales.

REFERENCES

[1] J. Schuhmann, R. Beaumont, R. Vencu, C. Gordon, C. Schubert, T. Brüggemann, A. Hosseini, A. Beitler, S. Chang, S. Cornillon, and others. LAION-5B: An open large-scale dataset for training next generation image-text models. *arXiv preprint arXiv:2210.08402*, 2022. DOI: https://doi.org/10.48550/arXiv.2210.08402

[2] Ansible Documentation. Ansible Automation Platform. Red Hat. Available: https://docs.ansible.com/

[3] SLURM Documentation. SLURM Workload Manager. Available: https://slurm.schedmd.com/documentation.html

[4] Prometheus Documentation. Prometheus Monitoring System. Available: https://prometheus.io/docs/

[5] T. Supersad. laion-distributed-pipeline: Distributed CLIP embedding and FAISS indexing system. GitHub Repository, 2025. Available: https://github.com/tylersupersad/laion-distributed-pipeline