MegaScale-Data: Scaling Dataloader for Multisource Large Foundation Model Training

Juntao Zhao[1,†] Qi Lu[2,†] Wei Jia[2,†] Borui Wan[1] Lei Zuo[2] Junda Feng[2]
Jianyu Jiang[2] Yangrui Chen[2] Shuaishuai Cao[2] Jialing He[2] Kaihua Jiang[2]
Yuanzhe Hu[2] Shibiao Nong[2] Yanghua Peng[2] Haibin Lin[2] Chuan Wu[1]

[1] *The University of Hong Kong* [2] *ByteDance Seed*

Abstract

Modern frameworks for training large foundation models (LFMs) employ dataloaders in a data-parallel manner, with each loader processing a disjoint subset of training data. When preparing data for LFM training that originates from multiple, distinct sources, two fundamental challenges arise. First, due to the quadratic computational complexity of the attention operator, the non-uniform sample distribution over data-parallel ranks leads to significant workload imbalance among dataloaders, degrading the training efficiency. Second, supporting diverse data sources requires per-dataset file access states that are redundantly replicated across parallel loaders, consuming excessive memory. This also hinders dynamic data mixing (e.g., curriculum learning) and causes redundant access/memory overhead in hybrid parallelism.

We present MegaScale-Data, an industrial-grade distributed data loading architecture for multisource LFMs training, with three key innovations: (1) Disaggregated data preprocessing via role-specific actors (Source Loaders/Data Constructors) to eliminate source and parallelism redundant data access and ensure multisource scalability. (2) Centralized and declarative data plane for load-time multisource orchestration, such as long-short context, multimodality, and curriculum learning. (3) Multi-level auto-partitioning and scaling mechanism for source loaders under heterogeneous preprocessing costs. We also contribute our designs and operational experience in deployment and fault tolerance. MegaScale-Data achieves up to: (1) 4.5× end-to-end training throughput improvement, and (2) 13.5× reduction in CPU memory usage.

CCS Concepts: • **Computing methodologies → Distributed computing methodologies**; **Artificial intelligence**.

Keywords: Distributed systems, Multisource Training, Data Loading

†Equal contribution.

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3803568

ACM Reference Format:
Juntao Zhao, Qi Lu, Wei Jia, Borui Wan, Lei Zuo, Junda Feng, Jianyu Jiang, Yangrui Chen, Shuaishuai Cao, Jialing He, Kaihua Jiang, Yuanzhe Hu, Shibiao Nong, Yanghua Peng, Haibin Lin, and Chuan Wu. 2026. MegaScale-Data: Scaling Dataloader for Multisource Large Foundation Model Training. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3767295.3803568

1 Introduction

The rise of large language and vision models, aka large foundation models (LFM), has propelled numerous downstream applications to achieve remarkable performance. Training LFMs faces major challenges on model efficiency and data efficiency. Model efficiency is related to effectively distributing a massive model parameters among GPUs to maximize resource utilization and reduce resource waste. Significant advances in training framework design [37, 53, 55, 59, 77] and novel parallelism paradigms [34, 45, 46] have been made, largely addressing model efficiency. Data efficiency, on the other hand, ensures that sufficient resources (e.g., CPU cores and DRAM) are allocated to the data preprocessing pipeline to avoid input data stalls during training and underutilization of valuable GPU hours.

Modern training frameworks partition the global training batch across data-parallel dataloaders, with each loader tasked with loading and preprocessing its assigned data subset. Specifically, LFM training data stems from diverse multilingual and multimodal sources, such as text (Wikipedia, Common Crawl [24]), images (ImageNet [16], LAION [56]), and domain-specific collections [20]. To meet the needs of LFM training, engineers typically depend on ad-hoc scripts to blend these datasets into *data mixtures* [7, 20, 62], which combine data sources at specified ratios for different use cases. Although some prior works [5, 28, 29, 51, 63, 76] have delved into data efficiency in model training, multisource preprocessing in LFM introduces unique challenges.

Multisource Data Orchestration. Under non-uniform sample distributions from diverse sources, the quadratic computational complexity of the attention operator [64] introduces significant workload imbalances. For example, a complete sequence composed of 30-token and 70-token subsequences incurs 16% more computation than two 50-token subsequences. These imbalances are manifested both intra- and inter-microbatch [74], creating stragglers over data parallel ranks, exacerbating pipeline bubbles over pipeline stages, and prolonging training time. The data imbalance is compounded by non-uniform data distribution across modality modules. In vision-language models (VLMs) [68, 70, 71], for instance, image encoders process raw pixels while language backbones operate on fused image-patch and text tokens. In that way, modality data within a training batch renders different workloads across modules within the VLM.

Multisource Scalability. Constructing data mixtures from massive sources introduces fundamental memory constraints. *First*, each dataloader worker process maintains independent data file access states via dedicated per-source resources: separate socket connections, schemas, metadata structures, and I/O buffers (e.g., Parquet Row Group [2]). This architecture imposes linear memory overhead growth concerning the number of data sources—a critical limitation given that modern LFM training incorporates hundreds of data sources. *Second*, practitioners perform multimodal *transformations* (JPEG/RGB conversion, video keyframe extraction [4, 22]) at runtime to avoid inflated storage (up to 200× for OCR). Such heavyweight processing necessitates vertical scaling of worker numbers to prevent data stalls, and dramatically increases the memory pressure of the dataloader. Disparities across data sources exacerbate the situation. For instance, audio processing requires 4× more computation per output token than image decoding and 300× more than text tokenization. Analogous heterogeneity also occurs in unimodal contexts through variable resolutions in images/videos. This forces the loader worker number to be sized for the data source with the largest preprocessing cost, creating severe resource over-provisioning.

Further, the *data mixture* is often dynamic. For instance, curriculum learning strategies [61] require the data mixture to evolve during training, starting with conceptually "easier" samples and progressively increasing the proportion of more challenging data. Similarly, other methods adapt the data mix based on real-time training metrics like loss [36]. This demands a data preprocessing framework that can express arbitrary mixing schedules and scale efficiently with evolving dataset preprocessing costs. The complexity is compounded by the hybrid parallelism strategies essential for LFM training. In common setups like pipeline parallelism (PP) or context parallelism (CP), multiple GPUs (ranks) collaborate on the same batch of data. For example, in a pipeline, different GPUs form stages that process a batch sequentially; in context parallelism, different GPUs process different segments of the same long sequence. Without a coordinated delivery system, the default approach is for each GPU's process to run a separate, identical dataloader. This naive approach causes massive waste, as the same data is redundantly fetched, preprocessed, and stored in memory on multiple devices, consuming significant I/O bandwidth and memory.

Current dataloaders [5, 28, 29, 47, 51, 54] are ill-equipped for the complexities of LFM training, as they are primarily designed for **single-source, data-parallel** scenarios rather than **multisource, hybrid-parallel** workloads. This design mismatch leads to two critical failures. First, they lack effective multisource orchestration: trainer-colocated loaders prevent global coordination, while remote loading systems provide limited APIs for scheduling and mixing across heterogeneous data sources. Second, for these loaders, each parallelism-unaware dataloader instance is a heavyweight clone that must independently manage the data access for the entire set of M data sources and K parallel ranks (CP/PP), leading to severe memory overhead that scales with source diversity and parallelization size.

To address these limitations, we present MegaScale-Data, a disaggregated data processing framework designed for large-scale foundation model workloads that require two core capabilities: global data orchestration and multisource scalability. The key insights behind MegaScale-Data are the disaggregation of the multisource data preparation and the final delivery. It decomposes multisource preprocessing into specialized roles—dedicated Source Loaders for sample-level transformations (e.g., JPEG decoding) and aggregating Data Constructors for batch-level operations (e.g., tensor padding, splitting)—to eliminate redundant source-level and parallelism-related data access, also mitigating connectivity overhead. It introduces a programmable data plane with DGraph, a stateful dataflow graph tracking dependencies and lineage for each data source, and ClientPlaceTree, a hierarchical topology model enabling hybrid parallelism-aware scheduling, to support declarative cross-module multisource data strategy definition. It further employs source auto-partitioning and mixture-driven auto-scaling to dynamically optimize worker allocation for heterogeneous preprocessing demands.

Our core contributions are summarized below:

▹ **Disaggregated multisource preprocessing architecture**: We design a distributed actor-model-based preprocessing pipeline that eliminates redundant data access and memory overhead in LFM multisource data preprocessing (Sec. 3)

▹ **Declaritive Load-time Data Orchestration**: The DGraph and ClientPlaceTree abstractions enable hybrid parallelism-aware data orchestration with minimal user coding effort (Sec. 4).

▹ **Adaptive Multisource Scaling**: We introduce scalable algorithms that dynamically optimize CPU utilization for data preprocessing based on heterogeneous source preprocessing costs and evolving data mixing ratios (Sec. 5).

▹ On clusters of up to 4096 GPUs, MegaScale-Data improves multimodal VLM training throughput by 4.5× and reduces CPU usage by 13.5× versus data parallel baselines. (Sec. 7).

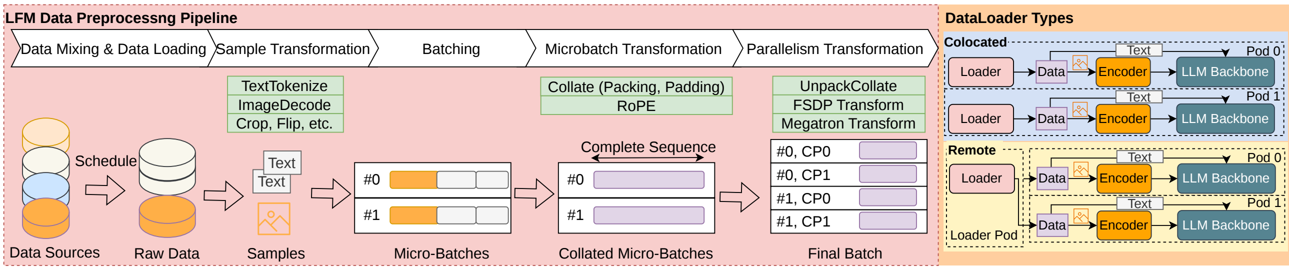


Figure 1. Left: Dataflow of Vision Language Model pretraining. Right: Two types of dataloaders.

2 Background and Motivation

We depict and analyze the production input data preprocessing pipeline for a Visual-Understanding-Language Model (VLM). Our analysis reveals the challenges and opportunities inherent in the data processing framework for LFM training.

2.1 LFM Input Data Preprocessing

As shown in Fig. 1, data from different datasets is mixed and loaded from cloud or distributed file systems, such as Amazon S3 [1] and HDFS [3]. On-the-fly transformations are conducted to convert input training samples into tensors that are ready for training. This "last-mile" data conversion is performed by online data processing frameworks, commonly referred to as dataloaders, such as the PyTorch DataLoader [54].

Data Mixing and Data Loading. In LFM training, samples from different data sources are combined in proportions to formulate a *data mixture* [62], resulting in a comprehensive, inclusive training corpus. The mixing ratio adjustment can be performed in a scheduled way, like with curriculum learning paradigms in reinforcement learning [61], warmup [42] and staged training [62] techniques, or in a dynamic manner, where the sampling weights of the data sources vary in response to evolving training status (such as loss and entropy [10, 36]). The dataloader utilizes the sampling ratios of the mixing schedule to load data proportionally from different datasets (sources).

Sample Transformations are applied to each data sample, converting the data format and improving the data quality [5, 29, 75]. For example, text tokenization and image decoding convert raw text and image data into trainable representations (e.g., text tokens and normalized RGB tensors). Similarly, the *crop* transformation standardizes image dimensions by cropping and resizing inputs to a fixed resolution.

Microbatch Transformation. After batching sampled data into microbatches, microbatch transformations collate samples to standardize the input shape. Packing merges fragmented subsequences into complete sequences with segmented masks [15, 40]. Padding aligns variable-length sequences by adding dummy tokens [15, 40]. Finally, positional embedding transformations like RoPE [11] are applied to obtain complete context information.

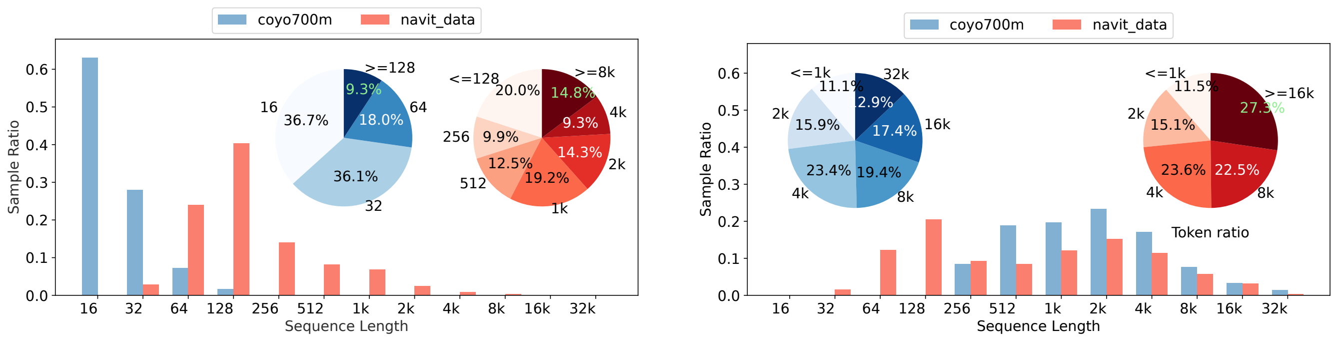
Parallelism Transformation. To effectively parallelize LFM training with increasing data and model sizes, hybrid parallelism strategies have been adopted. Hybrid parallelisms significantly influence how model training consumes training data. In data parallelism (DP) [14, 43, 58], microbatches are partitioned across training devices, and each device independently processes its local data using its model replica. Context parallelism (CP) [39, 46] partitions and scatters the input sequence; each device within a CP group consumes a part of the input sequence. Tensor parallelism (TP) [53] performs intra-operator partitioning, with each device within a TP group receiving the same input. Pipeline parallelism (PP) [33, 52] segments the model layers into stages, where only the first stage (PP0) obtains all microbatches, and intermediate results are exchanged between consecutive stages through peer-to-peer (P2P) communications. These replication and partitioning relationships between input data and hybrid parallelism schemes are encoded as parallelism transformations, applied after the microbatch transformation to ensure each client receives the correct input data.

Multimodal LFMs [32, 74] further complicate the input data processing pipeline with **heterogeneous colocation**. Multimodal LFMs are composed of more than one module (encoders, backbone, etc.), and each module may process different parts of the input data and employ different hybrid parallelism schemes. For example, a VLM can employ pure data parallelism for the vision encoder (e.g., ViT [19]) training and 4D (PP-DP-CP-TP) parallelism for LLM backbone (e.g., LLaMA [20, 48]) training. The parallelism difference for different data within a batch requires careful parallelism transformation programming.

2.2 Colocated and Remote Dataloader

As shown on the right of Fig. 1, conventional ML training frameworks (e.g., Torch [54]) typically colocate the dataloader with the training process. We observe that mainstream LFM training frameworks, such as Megatron-LM [53], DDP [44], FSDP [77], and MegaScale [37], still adhere to this practice. One advantage of colocated dataloaders is that they share the same sharding configuration (e.g., data parallelism) and CPU resources with the training process, eliminating the need for additional configuration. However, their rigid loader setup limits their ability to right-sizing resources like CPU and DRAM for efficient data processing.

Remote dataloaders, exemplified by Meta's Data Processing Service (DPP) [76] and Google's tf.data service [5], offload



(a) Text token distribution **(b)** Image token distribution

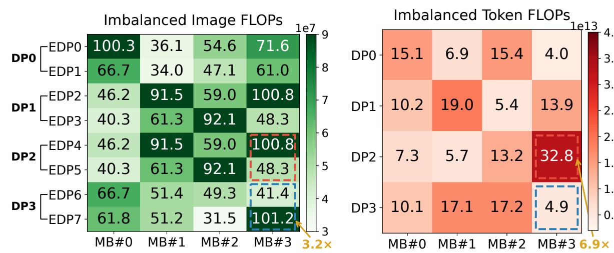
Figure 2. Skew token distribution in two datasets. Pie: total training token percentages by length; bar: sample ratios (e.g., short sequences dominate samples but long contribute more tokens). Left: Text token. Right: Image token.

data preprocessing to disaggregated CPU workers over the accelerator nodes. This architecture enables elastic scaling of data preprocessing throughput. Recent studies—Cachew [28], FastFlow [63], Cedar [75], and Pecan [29]—further optimize resource efficiency by orchestrating heterogeneous CPU resources across local and remote nodes. Nonetheless, all the existing dataloader systems fail to meet the new demands of LFM training, to be detailed in the following.

2.3 LFM Data Preprocessing Requirements

Multisource Data Orchestration. Data scheduling remains consistently necessary due to data heterogeneity under sophisticated training parallelism schemes. VLMs construct training samples by interleaving encoded images and text tokens. Each input sample comprises an image-text pair: images are split into patches and encoded into tokens via a visual encoder, while text labels are tokenized separately. These token streams are then interleaved to form complete training sequences. Fig. 2 shows the token distributions in the open-source *coyo700M* [9] dataset and our production *navit_data* dataset, where "text" represents text tokens, and "image" indicates the number of 16×16 (*coyo*) and 14×14 (*navit*) image patches [15]. Both distributions are significantly skewed. In *coyo700M*, 98.23% of samples contain text sequences $\leq$ 64 tokens, while the top 1.62% of longer sequences (> 64 tokens) account for 9.3% of all tokens. Similar skewness occurs in image subsequences. Such skewness manifests two critical computational challenges:

(1) Intra-module imbalance. The quadratic time complexity of the attention operation ($O(l^2)$ with l as sequence length) [13] induces a significant computational disparity between microbatches containing subsequences of varying lengths. Fig. 3 shows this disparity, where we benchmark an 8-card VLM training trial, colocating encoders and the backbone, employing Encoder Data Parallel (EDP) with size 8 to distribute image data across all GPUs, while the backbone utilizes hybrid parallelism with $DP = 4$ and $TP = 2$. The maximum microbatch FLOPs observed are 3.2× and 6.9× higher than



(a) Image flops heatmap **(b)** Token flops heatmap

Figure 3. Computational imbalance across microbatches. EDPx denotes the encoder data parallel rank x, DPx denotes the data parallel rank x, and MB#x represents the x-th microbatch.

the minimum for images and complete sequences, respectively, as indicated by the yellow arrows in Fig. 3.

(2) Inter-module imbalance. Input data distributions can vary significantly across modules. As shown in Fig. 3, even for the same microbatch, there exists a substantial divergence in the token distributions between raw image patches and the final collated sequence. Given the heterogeneous parallelisms of collocated modules on the same GPU, Data Parallel rank #3 (DP3) for the LLM backbone corresponds to Encoder Data Parallel ranks 6 and 7 (EDP6, EDP7). The computational complexity of image tokens across microbatch #3 is balanced between every two consecutive DP ranks (140e7); however, as highlighted by the red and blue dashed outlines, DP2's MB#3 workload (32.8×10^{13} FLOPs) is significantly larger than DP3's MB#3 (4.9×10^{13} FLOPs), indicating a severe imbalance. Moreover, when considering only the encoder data parallel ranks for MB#3, the FLOPs across DP ranks remain unbalanced and necessitate adjustment. To address this, data scheduling strategies should be tailored distinctly for different modules to homogenize their workloads.

Additionally, existing solutions often leverage all-to-all communication protocols at the model layer to aggregate information for load balancing input batches (e.g., [74]). However, this approach introduces four fundamental challenges:

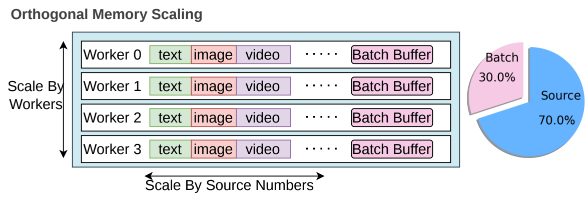
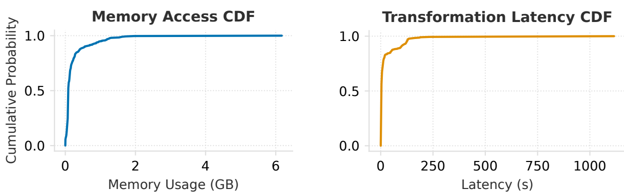


Figure 4. Orthogonal memory scaling by source and worker counts in multi-source preprocessing. Pie charts show that source-related memory dominates the usage.



(a) File access states memory **(b)** Transformation latency

Figure 5. Cumulative distribution function (CDF) of 100 samples across production source datasets.

increased activation size, elevated communication overhead, scalability bottlenecks, and tight coupling of load balancing logic with model-layer codebases, which risks contaminating the core computation workflow. We argue that load-time data orchestration presents a compelling alternative: an LFM data preprocessing system should proactively balance data from heterogeneous sources across modules before model ingestion.

Multisource Scalability. The multisource nature of LFM training data exerts substantial memory pressure on the preprocessing pipeline. As demonstrated in Fig. 4, our production-scale LFM training trials reveal that when the per-DP training batch size remains moderate, the memory footprint of replicated file access states for data sources constitutes over 70% of the total memory consumption during data preprocessing. This multisource memory overhead arises from two orthogonal scaling dimensions.

(1) Source Scaling. Modern LFMs achieve task generality through source datasets aggregated from diverse domains [7, 20, 72]. In production systems, training jobs process hundreds to thousands of distinct source files that collectively define the global data mixture. Each file introduces a fixed per-source memory overhead. For instance, practitioners commonly store training data in columnar formats such as Parquet [2], which optimize compression ratios and access locality through feature grouping [78]. Parquet files partition data into row groups (512MB–1GB storage units [2]). During reads, a client first establishes a dedicated socket to the file, loads metadata (e.g., footers, schema information) into memory, and executes queries over row groups using buffers [73]. This design inherently leads to linear memory overhead scaling: independent file states (sockets, metadata, buffers) per source cause memory usage to grow proportionally with the source number.

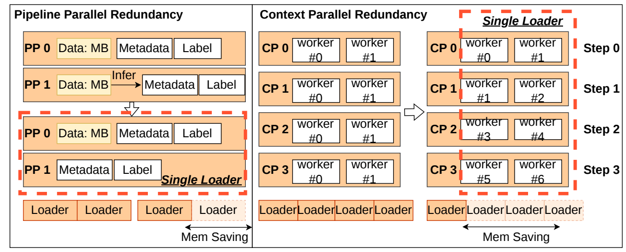


Figure 6. Optimizing Loader Parallelism Redundancy for Memory Saving. Left: share one loader instance result for per-stage loaders in the pipeline parallelism. Right: share one loader instance for per-rank loaders in context parallelism.

(2) Worker Scaling. To prevent GPU idling, practitioners must optimally size dataloader worker processes to match data transformation bottlenecks. Each worker process maintains its execution context and prefetch buffer [54], and thus the memory consumption scales with the number of workers. This issue is exacerbated by transformation time heterogeneity across modalities. For instance, text tokenization is lightweight, whereas image decoding (e.g., RGB conversion via PIL [12]) and video processing (e.g., keyframe extraction [4]) are computationally intensive. Even for the data of the same modality, e.g., images, patches [15] for variable-resolution images introduce order-of-magnitude cost differences. As a result, transformation latencies exhibit severe skewness across sources, as shown in Fig. 5. Critically, when faced with multiple data sources, loader workers must dynamically resize to match the throughput of the slowest transformation pipeline. This prevents vibrating feeding rates that could otherwise introduce GPU idle time. However, this forces dataloaders to scale workers based on worst-case latency demands, necessitating over-provisioning for faster pipelines.

These observations underscore the need for multisource-scalable preprocessing. An effective solution must address both (1) horizontal memory growth due to increasing sources and (2) vertical memory amplification from worker scaling under heterogeneous, time-varying transformation latency.

2.4 Challenges For Preprocessing Requirements

In addition to the above requirements, an efficient design of an LFM multisource data preprocessing framework faces the following challenges.

First, the dynamic data mixing of LFM training alters data source sampling ratios at runtime, necessitating the data orchestration panel to express and adapt to runtime data source sampling patterns. Additionally, it mandates that multisource scalability solutions not only support arbitrary mixing strategies but also automatically adjust the source mixing ratios as training progresses, given that per-source transformation latency evolves non-uniformly over time.

Next, as elaborated in Sec. 2.1, ranks in context parallelism and pipeline parallelism require only partial input data for model execution. In the absence of coordination, each rank independently instantiates a full dataloader to load complete batches for data partitioning and metadata retrieval (Fig. 6). This leads to redundant loader instances that scale with the number of CP/PP ranks, exacerbating memory overhead. To mitigate this, the framework must tightly integrate data orchestration with hybrid parallelism, eliminating redundant data access and memory occupation by sharing ephemeral, parallelism-aware preprocessed samples across ranks.

Finally, LFM training may dynamically adjust GPU allocations at runtime—elastic scaling (addition/deletion), redeployment, resharding, or failure events [65, 66]—which requires persistent data services to maintain stable data feeding speeds. These scenarios demand built-in mechanisms in the data proprocessing framework for resilient connectivity and fault tolerance to ensure uninterrupted data delivery.

2.5 Opportunities

Despite their advances, existing data loading systems are ill-equipped for the unique demands of large-scale, multi-source LFM training. The core optimizations offered by state-of-the-art remote dataloaders [28, 29] are fundamentally misaligned with multi-source LFM workloads. Their focus on caching offers little benefit in typical single-epoch training scenarios, and their emphasis on offloading CPU-bound transformations is less critical given the long training iteration time and abundant host CPU cores.

Consequently, these systems fail to address the primary bottlenecks in this domain: 1) the lack of expressive APIs to orchestrate the complex, dynamic data mixing required by hybrid parallelism, which forces developers into manual, error-prone implementations; and 2) severe memory redundancy that scales linearly with the number of concurrent data sources and loader instances, resulting in unsustainable resource costs. Furthermore, most systems lack robust fault tolerance, making large-scale training runs fragile.

MegaScale-Data addresses these issues with three architectural innovations. It employs an actor-model architecture to split preprocessing into Source Loaders and Data Constructors, improving scalability and eliminating redundant access in hybrid parallelism and multisource setups. A centralized data plane with declarative interfaces simplifies cross-module data scheduling. Multilevel source auto-partitioning and mixture-driven scaling optimize resource use. MegaScale-Data also ensures uninterrupted data service by incorporating shadow dataloaders for high-availability fault tolerance and an elastic resharding mechanism to dynamically adapt to training parallelism changes.

3 MegaScale-Data Overview

MegaScale-Data is a data preprocessing framework designed for unified orchestration and resource-efficient delivery of multisource training data. As shown in Fig. 7, the framework employs a hierarchical architecture with three core components, explicitly designed to address the scalability and orchestration challenges:

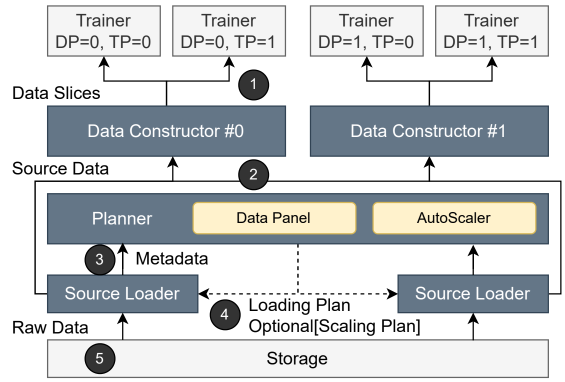


Figure 7. Architecture of MegaScale-Data. Solid lines denote data paths; dashed lines represent control paths.

Source Loader (Addressing Source Redundancy & Source Heterogeneity). To resolve memory constraints caused by source scaling, Source Loaders function as dedicated actors for specific data sources. By continuously ingesting data and applying sample-level transformations within isolated processes, they decouple file access states from worker processes, effectively eliminating the redundancy of file handles and metadata across the system.

Data Constructor (Addressing Parallelism Redundancy). Acting as the data sink for ranks (e.g., Data Parallel group), the Data Constructor aggregates outputs from Source Loaders to perform batch-level and parallelism transformations. This design enables seamless data sharing: ranks in the same CP group can generate partitions from identical batches, while PP stages can selectively exclude unnecessary metadata, significantly reducing memory and I/O overhead. The Larger cluster with inter-node parallelism requires finer-granular Data Constructor and selective broadcasting (See Sec. 6) to reduce synchronization cost.

Planner (Addressing Unified Orchestration & Scaling). The Planner orchestrates system-wide behavior through two key functions: (1) Plan Generation, which synthesizes loading plans by aggregating lightweight metadata (e.g., sample indices, sequence lengths) from Source Loaders and applying user-defined orchestration strategies; and (2) Auto-Scaling, which triggers resource resizing (Sec. 5) to adapt to preprocessing workload fluctuations and cost-efficiency targets.

Deployment. MegaScale-Data's components are deployed entirely in CPU memory. At bootstrap, the user defines a strategy using the MegaScale-Data programming model (Sec. 4), which the Planner's Data Panel then instantiates. This strategy provides policy-driven control over dataflows. Based on the trainer device topology, MegaScale-Data provisions Data Constructors and trainer clients. It also automatically partitions source datasets into multiple Source Loaders during instantiation, using journalized profiling results. To adapt to dynamic workload changes, the AutoScaler adjusts resource allocation in response to fluctuating mixing ratios (Sec. 5).

Workflow. The runtime data preprocessing follows a pull execution model [25, 27, 60]. Each component (Data Constructor, Source Loader, Planner) maintains its own task queue. The workflow proceeds as: ❶ A trainer-side client requests data from its Data Constructor. ❷ The Data Constructor triggers positional data fetches from all Source Loaders. ❸ Source Loaders consult the Planner to generate new loading plans. ❹ The Planner synthesizes new plans by collecting buffer metadata (e.g., sample indices, source signatures, sequence length) from Source Loaders. Finalized plans direct Source Loaders to prepare samples, pop them from the read buffers, stage them in queues, and signal a resource scaling plan if necessary. ❺ Source Loader reads new samples from the distributed storage and populates them into the buffer.

4 Data Orchestration Panel

4.1 Abstraction

Existing dataloader frameworks like tf.data.service [5] provide distributed execution APIs but lack native abstractions for two critical requirements: (1) multisource representation for data within the same batch, and (2) trainer-side hybrid parallelism scheme integration (e.g., 4D parallelism, PP, DP, CP, and TP). These gaps complicate unified implementation of multisource and cross-module data orchestration.

To address these limitations, MegaScale-Data introduces two core abstractions: *DGraph* for source-aware data transition state tracking and *ClientPlaceTree* for parallelism strategy resolution, as exemplified in Fig. 8.

DGraph is a state-tracking dataflow graph that models the lifecycle of training samples through explicit producer-consumer relationships. It is initialized by binding data samples to their respective Source Loaders. Upon plan generation, it associates each sample to its target Data Constructor. Each node in *DGraph* represents a training sample in a specific processing state, with directed acyclic edges encoding either data transformations or logical dependencies (e.g., microbatch grouping). Edges may be null if no state mutation occurs. *DGraph* operates on lightweight metadata and delivers two core advantages: *Unified multisource representation*, that allows creation of multiple source-specific graphs (e.g., test-image pair or text data from a specific source) from the same shared data dictionary through selective metadata specification; *Orchestration transparency*, which visualizes dataflow states, dependencies, and transformations via directed acyclic edges in an interpretable way.

ClientPlaceTree is a logical representation of the trainer device mesh. It provides a clear view of how data is accessed by trainers at different ranks and in different communication groups while abstracting details such as device memory capacities from users. Presenting the topology as a tree allows for easy modifications to the structure and automates the generation of parallelism transformation. The cost of rebuilding the ClientPlaceTree is negligible; any changes to the device mesh parallelism [65] are dynamically reflected in the loading plan computation, as detailed in Sec. 6.1. While typically inferred from the training configuration, the ClientPlaceTree allows users to override the default construction logic to implement custom behaviors, such as selective broadcasting in Sec. 6.

4.2 Primitives

As illustrated in Fig. 8, MegaScale-Data initializes its data orchestration primitives using buffer metadata collected from Source Loader buffers and a *ClientPlaceTree* encoding GPU allocations. Orchestration begins by specifying the modality metadata to create *DGraph*, enabling MegaScale-Data to encode either single or multiple modalities for load balancing. This modular approach is particularly valuable for models like VLMs, where multiple modules within the same process handle different data modalities at runtime, allowing separate specifications of balancing and transformation strategies. To facilitate this, MegaScale-Data adopts a declarative interface. We define several primitives for *DGraph* that allow users to describe the desired data distribution and transformation logic without managing low-level execution steps. These primitives can express most of our existing data orchestration strategies.

`mix(schedule)` enables real-time source mixing through scheduled sampling. Users define a multisource schedule that generates the sampling weight of source datasets for each training step (at epoch, step, or substep granularity [57]), determining the probabilistic selection of source data batches. Only sampled data participates in subsequent orchestration.

`distribute(axis, group_size)` specifies the axis along which data distribution occurs. This enables straightforward implementation of diverse partitioning strategies: (1) `axis='DP'` partitions data into minibatches across data-parallel groups; (2) `axis='CP'` treats $DP{\times}CP$ GPUs as uniform consumers for hybrid data parallelism [26]; (3) `axis='WORLD'` distributes data across all ranks for the encoder module in VLMs with world-wide data parallelism. Once the distribution axis is selected, MegaScale-Data automatically determines the parallelism transformations (e.g., CP transformations). The primitive creates n buckets corresponding to nodes at the specified axis level in the ClientPlaceTree hierarchy. When `group_size` is provided, the effective bucket count scales to $\lceil \frac{n}{\texttt{group\_size}} \rceil$, and samples are balanced within subgroups instead, reducing coordination overhead in super large clusters.

`cost(costfn)` registers a cost function that estimates compute and memory overhead from sample metadata. Costs are automatically propagated to subsequent `balance` operations.

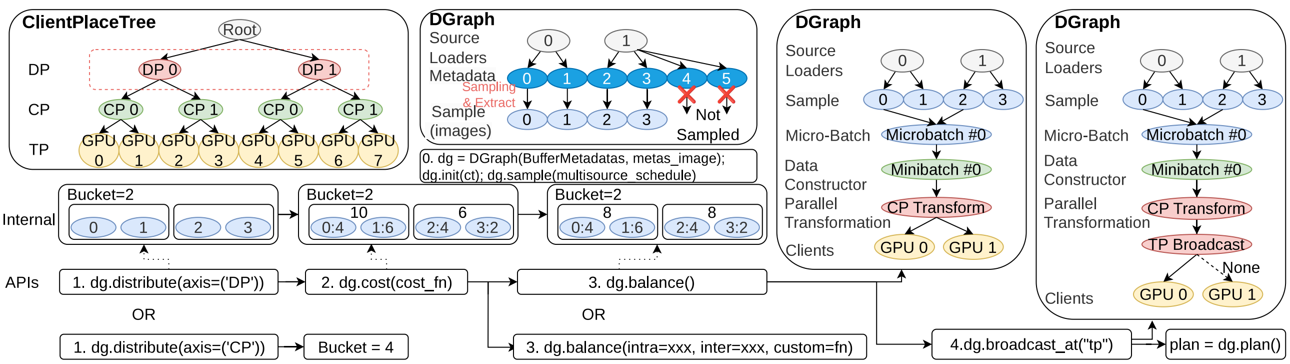


Figure 8. Multi-source data balancing under hybrid parallelism (DP=2, CP=2, TP=2). (Top) *DGraph* tracks sample lifecycles while *ClientPlaceTree* defines the logical device mesh. **(Middle)** *Internal* state: ellipses represent samples (e.g., "0:4" denotes ID 0 with cost 4); rounded rectangles are microbatch bins with aggregated costs shown above. **(Bottom)** API workflow: `distribute` creates n buckets based on the tree level (e.g., $n = 4$ for `axis='CP'`); `cost` maps metadata to overhead; `balance` redistributes samples into bins. Data (4, 5) are excluded based on the sampling results.

Specifically, we model the encoder's cost as a function of the image sequence length, the dimensions of the embedding and MLP layers, and the model's depth. The cost for the language backbone is likewise modeled as a function of the total sequence length and key architectural parameters, such as the number of experts per token, vocabulary size, and hidden layer dimensions. We validate the latency fidelity of our experimental cost models in Sec. 7.5.

`balance(method, *)` balances samples based on their computed costs. It further divides buckets into m bins, where m is the number of microbatches, and then applies the specified balancing method to distribute samples among these bins. We provide two candidate balancing methods: `greedybinpacking` and `karmarkar-karp` [8]. These can be applied at both intra-microbatch (bin) and inter-microbatch (bucket) levels [74], or interleaving. To keep the global batch unchanged, users may optionally disable intra-microbatch reordering via configuration. User-defined balancing strategies (e.g., Zig-Zag, V-Shape, and other interleaved patterns) can be implemented via the framework's extension APIs.

`broadcast_at(target_dim)` indicates to *DGraph* that a broadcast operation exists on the trainer side along the specified dimension. For example, when TP0 broadcasts data to all TP ranks, this directive informs the Data Constructor to exclude tp > 0 clients from data fetching. This mechanism optimizes communication by preventing redundant data access.

`plan()` dynamically generates optimized data-loading plans. Based on the plans, the Source Loader operates according to the plan to execute data mixing and scheduling, and then forwards prepared samples to the Data Constructor. The Data Constructor handles microbatch assembly and performs microbatch and parallelism transformations.

In addition to these primitives, we offer low-level programming interfaces with better flexibility, including `plan_raw`, `loader_do_plan`, `constructor_do_plan` and `summary_buffer`, which enable the user to program Planner, Source Loaders, and Data Constructor data consuming policies directly.

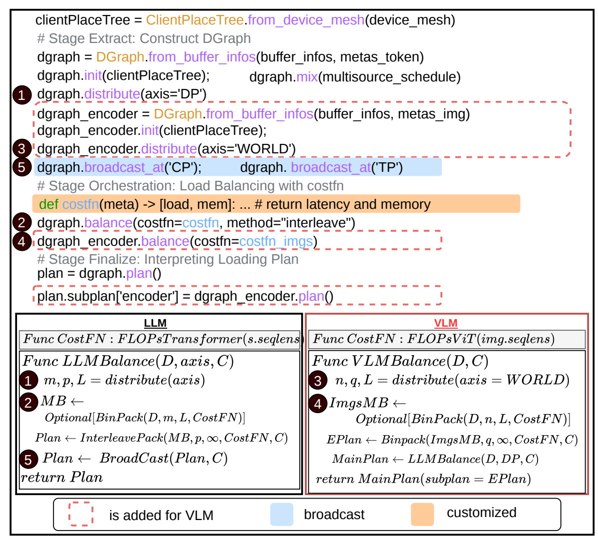


Figure 9. Strategy implementation of unimodal long-short sequence and multimodal vision language model pretraining task. Users can adapt to new training tasks by adding new lines and customizing limited functions.

4.3 Use Cases

As shown in Fig. 9, our APIs simplify the development of data orchestration strategies through a three-stage workflow: Extract, Orchestrate, and Finalize. To showcase the broad applicability and adaptability of our programming model, we analyze two representative use cases. In the scenario of unimodal long-short sequence 4D-parallel training across data-parallel ranks, taking data D, and ClientPlaceTree C as inputs, our framework achieves intra-module balancing (LLMBalance) using only seven lines of code, incorporating a cost model (*CostFN*). ❶ First, it distributes data along the DP axis to determine the number of data-parallel partitions, per-DP microbatches, and memory limit L. ❷ Next, it generates a resource consumption plan through balancing with the interleaving balancing strategy. ❺ If broadcast operations are included, the consumers are tailored. For the VLM setup, an inter-module balancing strategy is implemented with just five additional lines of code. The image DGraph is inferred using the same buffer but different metadata. We first distribute ❸ and balance ❹ images with data parallelism, and then combine with *LLMBalance* to perform global balancing across the modules. These concise implementations demonstrate the expressiveness and versatility of our framework in diverse LFM training environments.

5 MultiSource AutoScaler

The AutoScaler in MegaScale-Data addresses two core multisource scalability challenges: (1) partitioning and balancing heterogeneous preprocessing workloads and memory footprints across data sources, and (2) scaling of dynamic source mixing ratios. As shown in Fig. 10, MegaScale-Data employs a two-phase approach: ❶ **Offline Source Auto-Partitioning:** Sources are partitioned with loader parallel schemes to obtain Source Loader configurations. ❷ **Online Mixture-Driven Scaling:** At runtime, MegaScale-Data's AutoScaler in Planner dynamically scales and reshards Source Loaders in response to mixing ratio shifts, keeping low data preprocessing cost and resource efficiency under fluctuating demand.

5.1 Source Auto-Partitioning

We first analyze the parallelism schemes for partitioning the Source Loader, then introduce our scalable solution and elaborate its underlying rationale.

Dataloader Parallelism Schemes. The end-to-end preprocessing cost is modeled as a tuple (P, T, M), where P denotes transformation latency, T represents orchestration and data transfer overhead, and M signifies memory footprint, comprising batch buffer (M_b) and per-source file access states (M_d). As shown in Fig. 10, we analyze three parallelism strategies to distribute costs across loaders: (1) *Worker Parallel* amortizes P by staggering execution. Parallel workers in a loader initiate transformation $s - 1$ steps ahead of their successors, where w is the total worker count, enabling concurrent execution. (2) *Source Parallel* reduces M_d by partitioning data sources across multiple Source Loaders. Each loader worker maintains fewer file access states, decreasing memory pressure from metadata and I/O channels. (3) *Data Parallel* reduces P and M_b by sharding training data across loaders [5, 29], whose degree is bounded by the batch size.

MultiLevel Source Partition Given heterogeneous transformation costs $\{P_1, ..., P_k\}$ and memory footprints $\{M_1, ..., M_k\}$ across k data sources, we partition each source by default into multiple data-parallel actors, each containing worker-parallel workers with varying counts. The partition algorithm proceeds in three stages. (1) Source Clustering: We first sort all data sources in descending order of transformation cost P_k and cluster them into G source clusters. (2) Resource Level Construction: Using the ratio of mean transformation costs between the smallest and largest clusters, we estimate the number of workers for each source within a cluster. Available resources are calculated by subtracting resource allocations for the Data Constructor (estimated via fixed batch size) and Planner from total system resources. These available resources are divided by the total number of workers to form worker resource blocks. To prevent invalid data parallelism for global batches and physical pod resource overcommitment, we set upper bounds w_{src} (per-source worker limit) and w_{actor} (per-actor worker limit), and then generate actor counts (loader data parallelism degree) and worker counts for each resource level. (3) Configuration Generation: We derive resource configurations for each source within clusters. When memory resources are insufficient for M_k, we adjust the number of source actors to satisfy memory constraints.

5.2 Mixture-Driven Scaling

A key architectural advantage of MegaScale-Data is the Planner's centralized control over data mixture sampling, providing global visibility into cross-source mixing ratios (Fig. 7). This enables predictive autoscaling: as sampling weights evolve, the Planner dynamically adjusts Source Loader resources based on projected demand. When a source's moving-average sampling weight exceeds a threshold for consecutive intervals, the Planner triggers the AutoScaler to: (1) create new Source Loader actors, (2) reshard data partitions live, and (3) integrate the scaled actors into the plan generation. Idle resources are reclaimed under declining demand.

5.3 Design Rationale

We default to assigning one source per Source Loader actor, trading moderate communication for resource overhead, for two reasons. First, arbitrary mixing schedules complicate source grouping: maintaining stochastic consistency is challenging when sources in the same group exhibit divergent weight trajectories. Second, dedicated per-source actors eliminate most scaling needs from mixture changes, stabilizing feeding rates. While grouping remains suitable for static schedules, users can enable it in step (1) of source partition.

Our focus on horizontal scaling via actor/worker counts utilizes underutilized CPU resources co-located with accelerators. Empirical studies [30] and our trials show 75% idle auxiliary CPU capacity under static allocations; actor-level management enables finer-grained control.

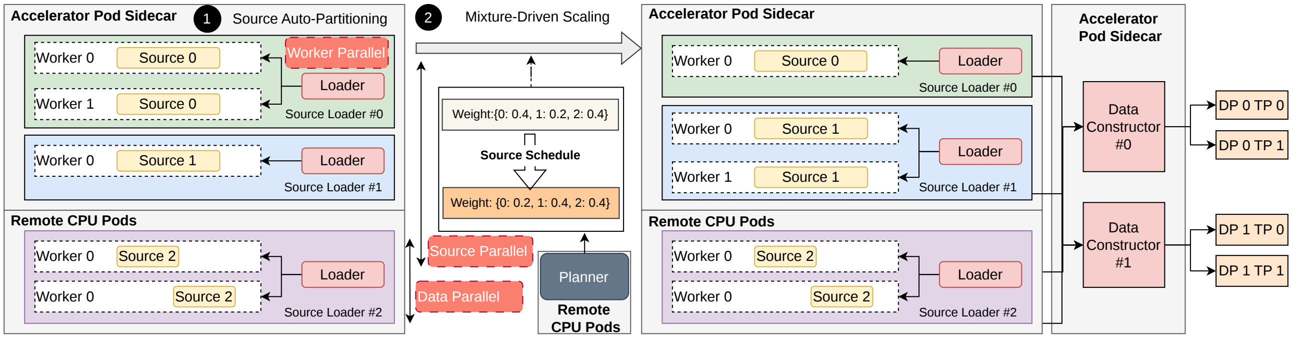


Figure 10. Two-Phase AutoScaling. (1) Offline Source Auto-Partitioning, (2) Online Mixture-Driven Scaling. The number of workers for each source loader is adjusted according to the transformation cost of the corresponding dataset.

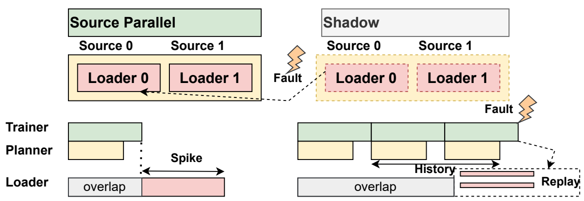


Figure 11. Failure Recovery: Shadow Loader, Replay with Differential Checkpointing Intervals.

6 Large-Scale Deployment

MegaScale-Data is implemented atop Ray [50], with 8763 lines of Python code. We discuss practical issues and solutions when deploying MegaScale-Data at scale.

6.1 Operational Adaptability

MegaScale-Data incorporates mechanisms for fault tolerance and elastic resharding to ensure continuous data service.

Fault Tolerance. We decouple recovery mechanisms by component role. Core coordinators (Planner, Data Constructor) leverage the Global Control Store (GCS) for state management and automatic restarts. MegaScale-Data uses persistent checkpointing and prefetch buffers to effectively mask this recovery latency. For Source Loaders, the Planner detects failures via RPC timeouts or payload integrity checks (e.g., partial yields lacking end-of-stream signals). As shown in Fig. 11, upon detection, it promotes hot-standby "Shadow Loaders" for instantaneous failover. To mitigate the latency of snapshotting large data buffers, we employ *differential checkpointing*: Source Loaders snapshot at a lower frequency than the Planner, utilizing deterministic replay of the Planner state to bridge the gap upon failure.

Elastic Resharding. To support dynamic changes in training parallelism, MegaScale-Data listens for notifications from the training framework. Upon receiving one, it immediately recalculates its data distribution plan for later incoming metadata and performs a fast resharding of resident data in Data Constructor to align with the new device topology.

Table 1. Model configurations.

	Model	#Layers	#Heads	Hidden Size
Encoder	ViT - 1B	39	16	1408
	ViT - 2B	48	16	1664
LLM	Llama - 12B	45	36	4608
	tMoE - 25B	42	16	2048 (topk = 2)
	Mixtral - 8×7B	32	32	4096 (topk = 2)

6.2 Deployment

We employ the following deployment tricks. **(1) Hybrid Deployment.** We adopt a hybrid model to balance efficiency and scalability (see Fig. 10). The Planner runs on remote CPU pods for centralized scheduling. In contrast, Source Loaders and Data Constructors are primarily deployed as *sidecars* within accelerator pods to utilize idle local CPU/memory resources [41]. They only scale out to remote CPU pods when sidecar resources become insufficient. **(2) Transformation Reordering.** Inspired by Pecan [29], we optionally defer image decoding to the Data Constructor, reducing communication data size. **(3) Selective Broadcasting.** To tackle the high synchronization overhead of trainer-side client barriers in large clusters, we propose bottom-up selective broadcasting over ClientPlaceTree (e.g., broadcasting tensors within sub-communication groups such as TP and CP). This increases memory and communication costs but reduces the number of synchronized clients.

7 Evaluation

We evaluate MegaScale-Data by quantifying its improvements in orchestration performance and multisource loading efficiency under large-scale deployment.

7.1 Experiment Setup

Models. We evaluate MegaScale-Data on Visual-Language Models (VLMs) using both dense and sparse Large Language Models (LLMs) as backbone, in combination with the Vision Transformer (ViT) [19] as the image encoder. The details of the models are given in Table 1. For the dense model, we

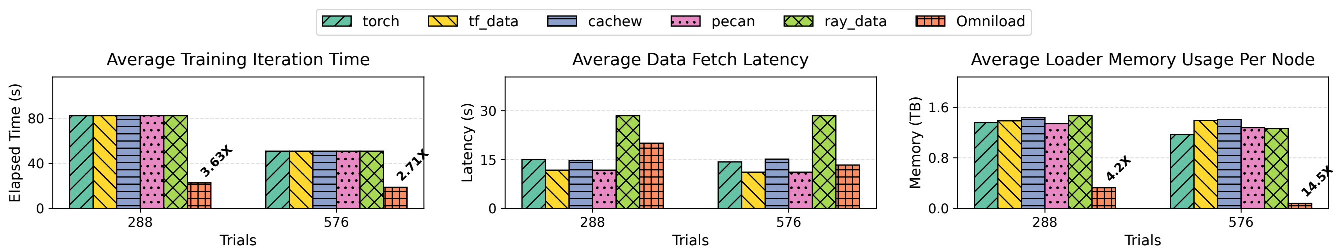


Figure 12. Comparison for data processing systems under different configurations. The parallelism strategies are ($TP = 4, PP = 8, DP = 9$) for the 288-GPU trial and ($TP = 4, PP = 4, CP = 4, DP = 9$) for the 576-GPU trial. For all cases, we make data fetch latency fully overlapped with training time, i.e., not input-bound.

select the Llama3 series [20]. For sparse models (Mixture-of-Experts - MoE), we choose the Mixtral series [35]. We also evaluate one production MoE model named tMoE.

Workloads. We use two dataset groups (Fig. 2 in Sec. 2.3), namely *coyo700m* and *navit_data*, consisting of 5 and 306 sources, respectively. We test MegaScale-Data across various batch sizes, context lengths, and GPU numbers. We further scaled up to 4096 GPUs to benchmark scalability in Sec. 7.5.

Baselines. We evaluate MegaScale-Data against state-of-the-art frameworks representing three data processing strategies: local processing (PyTorch DataLoader [54], tf.data [51]), remote processing (cachew [28], Ray Data [47]), and hybrid local-remote processing (Pecan [29]). For a fair comparison, we enable worker number auto-tuning for all data loaders to overlap data preprocessing time and model training iteration time with minimum CPU resources. By default, we enable the TP broadcasting for all data loaders.

For orchestration evaluation, we gauge the orchestration efficiency of our solution in three scenarios: (1) Vanilla, a baseline system without any data scheduling; (2) Backbone balance, which implements inter-microbatch load balancing exclusively on the LLM backbone; (3) Hybrid balance, which combines interleaved balancing (directly balancing sampled images across ranks) for the encoder with the backbone balance, as described in Fig. 9. We do not perform intra-microbatch balancing for the LLM backbone.

Metrics. We evaluate performance using three primary metrics: average iteration time (s), data fetch latency (s), and average loader memory usage per node (GB). For the orchestration evaluation, we also report training throughput in tokens per second (tokens/s). All results are collected over 100 training iterations following an initial warmup period. To ensure a fair comparison, the memory footprint of shadow loaders is excluded from all measurements.

Testbed. We conduct our experiments on a training cluster composed of multiple nodes, each equipped with 16× NVIDIA L20 GPUs (48GB) and 1.8TB of host DRAM. We employ the sidecar mode to launch isolated containers on the host, allocating half of the available CPU cores and memory resources to the resource pool for Ray scheduling. The entire training cluster is interconnected via InfiniBand, with HDFS as the storage backend.

7.2 Data Preprocessing Architecture Evaluation

We demonstrate resource savings achieved by MegaScale-Data using the Llama-12B + ViT-2B combination on 288 and 576 GPUs, with a batch size of 72 for each GPU. To utilize the GPU resources and avoid HBM OOM (out of memory), we size the backbone layer number to fit the model into the GPU memory. We restrict the model layers to 8 and 16, respectively. We measure the memory usage at each node and calculate the average memory per node.

As illustrated in Fig. 12, MegaScale-Data achieves up to a 3.63× speedup in training iteration time compared to state-of-the-art loaders, attributable to its efficient load-time data orchestration. It also delivers substantial resource savings, with up to a 13.5× reduction in loader memory usage. This efficiency stems from its ability to eliminate parallelism and source-level data redundancies that existing systems cannot address. While MegaScale-Data incurs a minor fetch overhead due to its coordination mechanisms, this latency is fully masked by the training computation. Notably, the reduced average per-node memory overhead in the 576-GPU configuration is attributed to the auto-partitioning feature, as more abundant CPU resources enable a larger optimization space for the same number of involved data sources.

7.3 Orchestration Evaluation

The results of the end-to-end orchestration performance are shown in Fig. 13. Compared to the non-scheduling baseline, our approach achieves up to 4.54× (average 1.77×) throughput improvement. The following key observations are made.

Larger Context Lengths Amplify Heterogeneity. Longer contexts also introduce greater in-batch heterogeneity. Our balancing strategy capitalizes on this property, yielding higher gains: 4k contexts achieve an average speedup of 1.71×, 8k contexts yield 2.63×, and 16k contexts reach 3.09×. In the absence of load balancing, peak activation memory can induce OOM errors, as empirically observed in our ViT-2B experiments. The results also validate the necessity of our load-time balancing strategy, which also mitigates the excessive memory overhead caused by all-to-all collective communication.

Dataset Characteristics Influence Gains. Fig. 2 shows that *coyo700m* contains shorter text subsequences than *navit_data*,

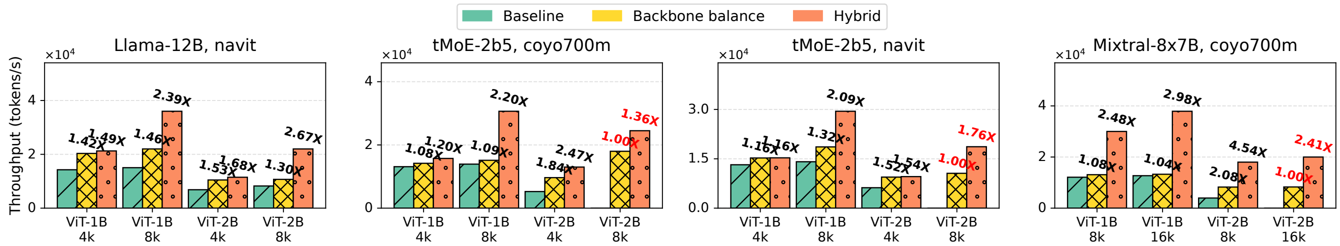


Figure 13. End-to-end orchestration performance across varying context lengths, dataset groups, and model sizes.

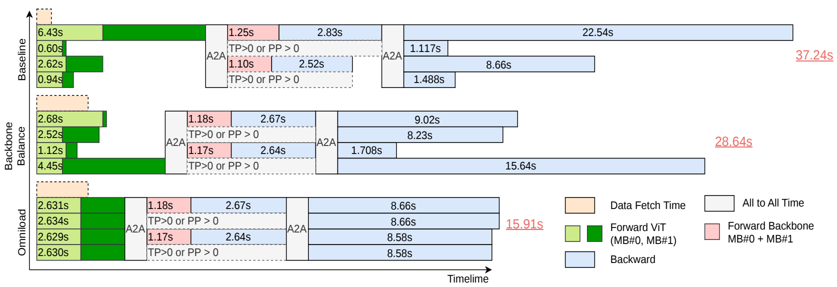


Figure 14. Case Study for VLM pre-training data orchestration.

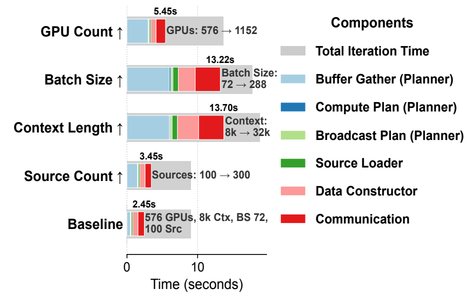


Figure 15. Time breakdown.

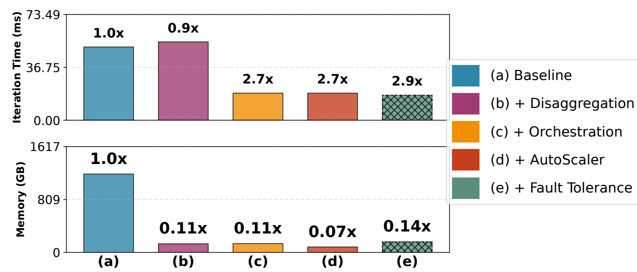


Figure 16. Components Contributions.

leading to greater heterogeneity for image computation with equivalent context lengths. This manifests higher speedups with hybrid balancer: *coyo700m* achieves 2.48× average (up to 4.54×) versus *navit_data*'s 2.42× average (up to 3.47×). For 4k contexts with *navit_data*, encoder balancing provides limited benefit as small context length results in few samples inside a batch, which offers very limited scheduling space.

Encoder Scaling Affects Strategy Efficacy. Hybrid balancing (encoder + backbone) outperforms encoder-only strategies more significantly with larger encoders. For Llama-12B, ViT-2B shows 1.58× gain versus ViT-1B's 1.41× under hybrid balancing. This observation is intuitive: the encoder serves as the bottleneck in our current model configuration, and a larger encoder enables more substantial benefits when hybrid balancing is applied.

7.4 Case studies

To demonstrate the benefits of MegaScale-Data, we present a case study using a Llama-12B and ViT-2B model on the *navit_data* dataset (Fig. 14). The experiment uses a batch size of 128, 8192 max sequence length, with a hybrid parallelism strategy (PP=9, DP=8, CP=2, TP=4), which requires an All-to-All communication collective to transfer features from the ViT encoder to the LLM backbone.

The baseline implementation suffers from a severe workload imbalance in the encoder stage, stemming from variable input image resolutions. This imbalance occurs because the maximum sequence length parameter only constrains the token count for the LLM backbone; it does not regulate the number of visual tokens produced by the encoder, which is dependent on image size. While a naive, microbatch-level balancing scheme is too coarse-grained to resolve this issue, MegaScale-Data integrates a hybrid balancer into the data loading process. This component proactively constructs batches to ensure an even workload distribution across all stages. By addressing the root cause of the imbalance, MegaScale-Data minimizes worker idle time, reducing the end-to-end iteration latency from 37.24s to 15.91s and achieving a 2.34× speedup.

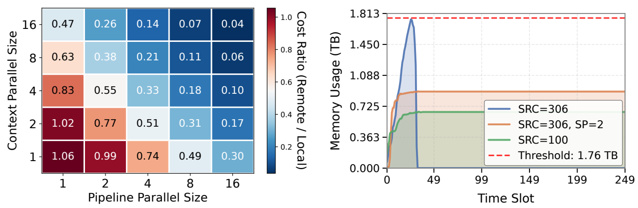
7.5 Ablation Studies

Time Breakdowns. Fig. 15 illustrates the time breakdown of MegaScale-Data components in training iteration. Across all configurations, including on a 1,152-GPU cluster, the data fetch overhead is minimal and consistently overlapped by the total training iteration time (gray bar).

The overhead scales gracefully with training configuration. Increasing the number of data sources only slightly impacts buffer gathering time. While larger sequence lengths and batch sizes (bounded by GPU HBM) increase overhead from data construction and planning, the training time scales commensurately. This proportional dynamic ensures the data pipeline overhead remains effectively hidden behind computation. Finally, we note that the Planner and Data Constructor are not yet heavily optimized, presenting opportunities for further latency reductions.

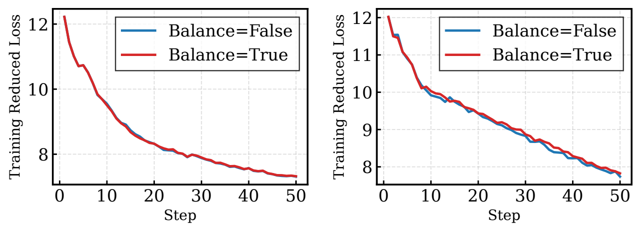
Table 2. API cost for data orchestration under different training setups. Baseline setup is Llama-12B + ViT-2B, 288 GPUs, BS=72, max sequence length (Seq) = 8k.

Case	Baseline	+BS 72→144	+Seq 8K→16K	+Cluster 288→1152	+Group 1→2, 1152 GPUs
API:*cost* (s)	0.004	0.006	0.012	0.107	0.106
API:*balance* (s)	0.016	0.027	0.173	0.357	0.195
Iteration Time (s)	14.31	15.98	16.91	13.56	13.56



(a) Parallelism redundancy. **(b)** Source redundancy.

Figure 17. Peak host memory usage when eliminating (a) parallelism redundancy and (b) source redundancy. SP=2 evenly partitions data sources across data-parallel ranks.



(a) Without CP. **(b)** With CP.

Figure 18. Impact of the balancer on training loss convergence. (a) Without Context Parallelism (CP), the balanced loss tightly tracks the baseline. (b) With CP, the balancer introduces minor fluctuations but maintains overall convergence.

Component Ablations. We conducted an ablation study to quantify the contribution of each component in MegaScale-Data (Fig. 16) in the 576-GPU experiment. Disaggregation significantly reduces loader memory usage by eliminating redundant loaders, at the cost of a 10% latency increase. Building on this, Orchestration achieves a 2.7× end-to-end performance speedup with negligible memory overhead. The AutoScaler then further optimizes memory utilization by dynamically adjusting resources. Finally, Fault Tolerance, using two shadow loaders, improves the effective training throughput rate (ETTR) by 1.08× during failures, in exchange for a predictable increase in memory footprint.

API Scalability. Table 2 shows negligible API costs for data orchestration in scaled Llama-12B + ViT-2B experiments on the *navit* dataset. Batch size and sequence length are bounded by GPU HBM capacity. Results confirm the API overhead is highly scalable and much smaller than training iteration time. Although API cost grows with cluster size, setting group size properly effectively controls this increase while maintaining orchestration performance.

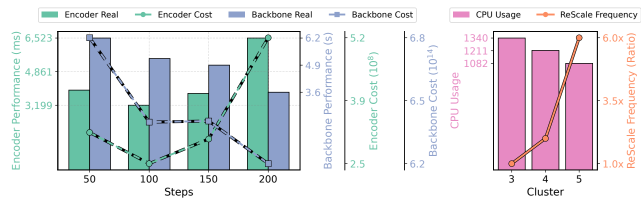


Figure 19. Cost model fidelity and clustering size impact.

Impact on Training Loss. Fig. 18 evaluates the impact of the balancer on training convergence across previous experimented trials: without and with Context Parallelism (CP). As detailed in Sec. 7.1, we adopt a conservative balancing strategy, where the hybrid balancer performs only inter-microbatch balancing across the backbone while preserving intra-microbatch sample randomness. As shown in Fig. 18a, in the absence of CP, the training loss of the balanced configuration tightly mirrors the non-balanced baseline. Conversely, when CP is enabled (Fig. 18b), the balanced loss exhibits slightly more fluctuation and minor deviations from the baseline, though it still converges effectively. We attribute this variance to the modified sequence partitioning induced by global balancing. This repartitioning alters the distribution of tokens across devices, which inherently introduces minor, acceptable numerical differences during distributed matrix multiplication (GEMM) and summation operations.

Redundancy Removal. We evaluate MegaScale-Data's design efficiency in reducing two primary sources of redundancy: high parallelism and duplicated data sources. First, to quantify parallelism redundancy, we profile memory usage on a simulated backend configured for a large-scale scenario (512 GPUs, BS=512) without source partitioning. As shown in Fig. 17a, memory efficiency gains become significant as context and pipeline parallelism levels increase. We then specifically evaluate the efficacy of source partitioning by isolating the data loader on a cluster configured with TP=16, worker=8 and DP=2. Fig. 17b reveals that uniformly partitioning data sources across DP ranks yields substantial reductions in memory overhead, demonstrating our system's comprehensive approach to optimizing memory.

Cost Model and AutoScaler Analysis. Fig. 19 validates our performance model and evaluates the trade-offs of auto-partitioning. The left panel validates our model's accuracy, showing that its predictions for the encoder and a single-layer backbone closely track empirical measurements. The right panel illustrates the trade-off between improved load balancing from more partitions and the resulting increase in rescaling costs. We identify a partition size of 4 as the optimal balance for the evaluated production workloads.

Scalability Advantages of the Actor Model To evaluate the scalability of our disaggregated structure (Fig. 20), we train a pure-text model, comparing MegaScale-Data against a direct-transfer baseline that bypasses the Data Constructor. While performance is comparable at 1k GPUs, scaling to 2k GPUs reveals severe connection overhead in the baseline,

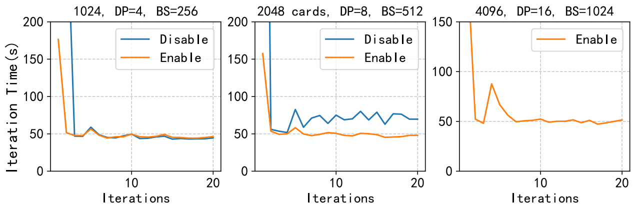


Figure 20. Scalability Advantages of the Actor Model.

which suffers a 10× data fetch latency increase compared to MegaScale-Data. At 4k GPUs, the baseline completely collapses due to communication bottlenecks. In contrast, MegaScale-Data sustains throughput via data redistribution, underscoring the Data Constructor's critical role in extreme-scale training.

8 Related Work

Remote dataloader. Disaggregated data loading architectures [5, 76] offload data preprocessing to remote workers to improve resource management and prevent data stalls. Subsequent systems introduced optimizations such as auto-caching [28], auto-scaling, selective worker placement [29], and combinatorial pipeline search [75]. However, these approaches are misaligned with LFM training, where single-epoch processing renders data heterogeneity and memory redundancy from numerous sources—not CPU limitations—is the primary bottleneck. In contrast, MegaScale-Data is specifically designed to eliminate this memory redundancy and provide the flexible, multisource orchestration required by LFMs.

Multimodal Training Frameworks. Recent frameworks seek to boost the efficiency of multimodal training by addressing model and data heterogeneity [23, 32, 74]. These systems primarily employ model-level optimizations, such as applying distinct parallelism to submodules [32], scheduling computations to fill pipeline bubbles [23], and disaggregating model components for independent resource management [74]. While effective at balancing the model's computational workload, these methods overlook opportunities in data orchestration. In contrast, MegaScale-Data introduces dynamic, load-time balancing through a disaggregated data architecture, optimizing performance without adding communication overhead to the critical path or increasing memory consumption.

Fault-tolerant training. Previous studies [18, 31, 38, 67, 79] have introduced robust AI infrastructures to automatically address various incidents during training. These systems employ a variety of techniques for failure detection and diagnosis [17, 18, 31, 38], failover procedure optimizations [6, 67], and checkpoint recovery [21, 49, 66, 69]. Unlike prior work, MegaScale-Data enhances the fault tolerance of the data loading module by providing differential checkpointing for its components and utilizing shadow loaders for hot recovery to ensure consistent data delivery during training.

9 Future Work

Replay Mode. While our system supports dynamic data mixing to adjust sampling ratios at runtime, many production training workloads employ predictable learning schedules. For such cases, the orchestration plan for each step can be pre-computed offline. By decoupling planning from execution, the runtime can transition to a "Replay Mode" that executes pre-computed schedules, significantly reducing online Planner overhead. This architectural shift allows the Planner to prioritize high-level health monitoring and fault tolerance over per-step data orchestration.

Ahead-of-Fetch Load Balancing. Current source loaders perform load balancing reactively after fetching data from the storage. We aim to explore Ahead-of-Fetch balancing by leveraging early metadata acquisition (e.g., pre-computing sample costs and embedding them directly into the storage). This design simultaneously enhances architectural flexibility and system efficiency: by decoupling metadata retrieval from the source loader, it allows for more flexible transformation placement within the data preprocessing pipeline and creates potential cost savings.

Strategy Optimizer. Since our data plane is declarative, we plan to develop an optimizer that automatically rewrites and fuses orchestration strategies. Specifically, we aim to explore graph rewriting techniques to fuse orchestration primitives and translate the execution plan into efficient, low-latency C kernels.

10 Conclusion

MegaScale-Data tackles the challenge of scalable multi-source data preprocessing and orchestration for large foundation model training. The actor-model preprocessing framework eradicates source and parallelism redundancy in multisource data loading, ensuring scalable preprocessing for LFM training jobs. The declarative data plane simplifies the programming of intricate cross-module, multi-source data scheduling and mixture sampling strategies. Leveraging source autoscaling, we partition and organize heterogeneous sources effectively. Experiments show that MegaScale-Data outperforms state-of-the-art baselines significantly in terms of throughput and resource efficiency, while maintaining scalability and robustness.

Acknowledgment

We would like to thank the anonymous EuroSys Reviewers and our shepherd, Bo Zhao, for their insightful feedback. This work was supported in part by a ByteDance Research Collaboration Project, and grants from Hong Kong RGC under the contracts HKU 17204423, 17205824, 17204625, C7004-22G (CRF), and T43-513/23-N (TRS).

References

[1] Amazon Web Services. 2025. Amazon S3 (Simple Storage Service). https://docs.aws.amazon.com/zh_cn/emr/latest/ReleaseGuide/emr-hbase-s3.html Accessed: 2025-03-22.

[2] Apache Software Foundation. 2025. Apache Parquet Documentation: File Format Configurations. https://parquet.apache.org/docs/file-format/configurations/ Accessed: 2025-03-22.

[3] Apache Software Foundation. 2025. Hadoop Distributed File System (HDFS). https://docs.aws.amazon.com/zh_cn/emr/latest/ReleaseGuide/emr-encryption-tdehdfs.html Accessed: 2025-03-22.

[4] Milan K Asha Paul, Jeyaraman Kavitha, and P Arockia Jansi Rani. 2018. Key-frame extraction techniques: A review. *Recent Patents on Computer Science* 11, 1 (2018), 3–16.

[5] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiří Šimša, and Chandramohan A. Thekkath. 2023. tf.data service: A Case for Disaggregating ML Input Data Processing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 358–375.

[6] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent Shafey, Chandu Thekkath, and Yonghui Wu. 2022. Pathways: Asynchronous Distributed Dataflow for ML. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. MLSys Organization, Santa Clara, CA, USA, 430–449.

[7] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. arXiv:2401.02954 [cs.CL] https://arxiv.org/abs/2401.02954

[8] S. Boettcher and S. Mertens. 2008. Analysis of the Karmarkar-Karp differencing algorithm. *The European Physical Journal B* 65, 1 (Aug. 2008), 131–140. https://doi.org/10.1140/epjb/e2008-00320-9

[9] Minwoo Byeon, Beomhee Park, Haecheon Kim, Sungjun Lee, Woonhyuk Baek, and Saehoon Kim. 2022. COYO-700M: Image-Text Pair Dataset. https://github.com/kakaobrain/coyo-dataset.

[10] Mayee F. Chen, Nicholas Roberts, Kush Bhatia, Jue Wang, Ce Zhang, Frederic Sala, and Christopher Ré. 2023. Skill-it! A Data-Driven Skills Framework for Understanding and Training Language Models. arXiv:2307.14430 [cs.CL] https://arxiv.org/abs/2307.14430

[11] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending Context Window of Large Language Models via Positional Interpolation. arXiv:2306.15595 [cs.CL] https://arxiv.org/abs/2306.15595

[12] Clark C. Evans and contributors. 2024. Pillow Library. https://pillow.readthedocs.io/en/stable/. Python Imaging Library (PIL) Fork.

[13] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*. Curran Associates Inc., Red Hook, NY, USA, 16 pages.

[14] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc.

[15] Mostafa Dehghani, Basil Mustafa, Josip Djolonga, Jonathan Heek, Matthias Minderer, Mathilde Caron, Andreas Steiner, Joan Puigcerver, Robert Geirhos, Ibrahim M Alabdulmohsin, Avital Oliver, Piotr Padlewski, Alexey Gritsenko, Mario Lucic, and Neil Houlsby. 2023. Patch n' Pack: NaViT, a Vision Transformer for any Aspect Ratio and Resolution. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 2252–2274. https://proceedings.neurips.cc/paper_files/paper/2023/file/06ea400b9b7cfce6428ec27a371632eb-Paper-Conference.pdf

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255.

[17] Yangtao Deng, Lei Zhang, Qinlong Wang, Xiaoyun Zhi, Xinlei Zhang, Zhuo Jiang, Haohan Xu, Lei Wang, Zuquan Song, Gaohong Liu, et al. 2025. Mycroft: Tracing Dependencies in Collective Communication Towards Reliable LLM Training. *arXiv preprint arXiv:2509.03018* (2025).

[18] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, Zhihui Ren, Zhicheng Wang, Bin Luo, Peng Zhang, Yang Liu, Yanqing Chen, Yu Guan, Weicheng Wang, Chaojie Yang, Yang Zhang, Man Yuan, Hanyu Zhao, Yong Li, Zihan Zhao, Shan Li, Xianlong Zeng, Zhiping Yao, Binzhang Fu, Ennan Zhai, Wei Lin, Chao Wang, and Dennis Cai. 2025. Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 865–881. https://www.usenix.org/conference/nsdi25/presentation/dong

[19] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.

[20] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. arXiv:2407.21783 [cs.AI]

[21] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 929–943. https://www.usenix.org/conference/nsdi22/presentation/eisenman

[22] Haoqi Fan, Tullie Murrell, Heng Wang, Kalyan Vasudev Alwala, Yanghao Li, Yilei Li, Bo Xiong, Nikhila Ravi, Meng Li, Haichuan Yang, Jitendra Malik, Ross Girshick, Matt Feiszli, Aaron Adcock, Wan-Yen Lo, and Christoph Feichtenhofer. 2021. PyTorchVideo: A Deep Learning Library for Video Understanding. In *Proceedings of the 29th ACM International Conference on Multimedia* (Virtual Event, China) *(MM '21)*. Association for Computing Machinery, New York, NY, USA, 3783–3786. https://doi.org/10.1145/3474085.3478329

[23] Weiqi Feng, Yangrui Chen, Shaoyu Wang, Yanghua Peng, Haibin Lin, and Minlan Yu. 2024. Optimus: Accelerating Large-Scale Multi-Modal LLM Training by Bubble Exploitation. arXiv:2408.03505 [cs.CL] https://arxiv.org/abs/2408.03505

[24] Common Crawl Foundation. 2014. Common Crawl. https://commoncrawl.org.

[25] Diego García-Gil, Sergio Ramírez-Gallego, Salvador García, and Francisco Herrera. 2017. A comparison on scalability for batch big data processing on Apache Spark and Apache Flink. *Big Data Analytics* 2 (2017), 1–11.

[26] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. 2025. ByteScale: Efficient Scaling of LLM Training with a 2048K Context Length on More Than 12,000 GPUs. arXiv:2502.21231 [cs.DC] https://arxiv.org/abs/2502.21231

[27] G. Graefe. 1994. Volcano: An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (Feb. 1994), 120–135. https://doi.org/10.1109/69.273032

[28] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. 2022. Cachew: Machine Learning

Input Data Processing as a Service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 689–706.

[29] Dan Graur, Oto Mraz, Muyu Li, Sepehr Pourghannad, Chandramohan A. Thekkath, and Ana Klimovic. 2024. Pecan: Cost-Efficient ML Data Preprocessing with Automatic Transformation Ordering and Hybrid Placement. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 649–665. https://www.usenix.org/conference/atc24/presentation/graur

[30] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. 2024. Characterization of Large Language Model Development in the Datacenter. arXiv:2403.07648 [cs.DC] https://arxiv.org/abs/2403.07648

[31] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. 2024. Characterization of Large Language Model Development in the Datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 709–729. https://www.usenix.org/conference/nsdi24/presentation/hu

[32] Jun Huang, Zhen Zhang, Shuai Zheng, Feng Qin, and Yida Wang. 2024. DISTMM: accelerating distributed multimodal model training. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'24)*. USENIX Association, USA, Article 64, 15 pages.

[33] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. *GPipe: efficient training of giant neural networks using pipeline parallelism*. Curran Associates Inc., Red Hook, NY, USA.

[34] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Reza Yazdani Aminadabi, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2024. System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing* (Nantes, France) *(PODC '24)*. Association for Computing Machinery, New York, NY, USA, 121–130.

[35] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] https://arxiv.org/abs/2401.04088

[36] Yiding Jiang, Allan Zhou, Zhili Feng, Sadhika Malladi, and J. Zico Kolter. 2024. Adaptive Data Optimization: Dynamic Sample Selection with Scaling Laws. arXiv:2410.11820 [cs.LG] https://arxiv.org/abs/2410.11820

[37] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, USA, 745–760.

[38] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[39] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. arXiv:2205.05198 [cs.LG] https://arxiv.org/abs/2205.05198

[40] Mario Michael Krell, Matej Kosec, Sergio P. Perez, and Andrew Fitzgibbon. 2022. Efficient Sequence Packing without Cross-contamination: Accelerating Large Language Models without Impacting Performance. arXiv:2107.02027 [cs.CL] https://arxiv.org/abs/2107.02027

[41] Kubernetes. 2024. Sidecar Containers. https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/ Kubernetes Documentation v1.29.

[42] Conglong Li, Minjia Zhang, and Yuxiong He. 2022. The Stability-Efficiency Dilemma: Investigating Sequence Length Warmup for Training GPT Models. arXiv:2108.06084 [cs.LG] https://arxiv.org/abs/2108.06084

[43] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. arXiv:2006.15704 [cs.DC] https://arxiv.org/abs/2006.15704

[44] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3005–3018.

[45] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, Shen Li, Zhigang Ji, Tao Xie, Yong Li, and Wei Lin. 2024. Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache. arXiv:2401.02669 [cs.DC] https://arxiv.org/abs/2401.02669

[46] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context.

[47] Frank Sifei Luan, Ziming Mao, Ron Yifeng Wang, Charlotte Lin, Amog Kamsetty, Hao Chen, Cheng Su, Balaji Veeramani, Scott Lee, SangBin Cho, Clark Zinzow, Eric Liang, Ion Stoica, and Stephanie Wang. 2025. The Streaming Batch Model for Efficient and Fault-Tolerant Heterogeneous Execution. arXiv:2501.12407 [cs.DC] https://arxiv.org/abs/2501.12407

[48] Meta AI. 2025. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/ Accessed: 2025-04-06.

[49] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 203–216. https://www.usenix.org/conference/fast21/presentation/mohan

[50] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: a distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 561–577.

[51] Derek G. Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A Machine Learning Data Processing Framework. arXiv:2101.12127 [cs.LG] https://arxiv.org/abs/2101.12127

[52] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM symposium on operating systems principles*. Association for Computing Machinery, New York, NY, USA, 1–15.

[53] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, 1–15.

[54] PyTorch contributors. 2024. torch.utils.data — PyTorch 2.4 documentation. https://pytorch.org/docs/stable/data.html Accessed: [Insert access date].

[55] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Association for Computing Machinery, New York, NY, USA, 3505–3506.

[56] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. 2021. LAION-400M: Open Dataset of CLIP-Filtered 400 Million Image-Text Pairs. arXiv:2111.02114 [cs.CV]

[57] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG]

[58] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799 [cs.LG] https://arxiv.org/abs/1802.05799

[59] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL]

[60] Rishika Shree, Tanupriya Choudhury, Subhash Chand Gupta, and Praveen Kumar. 2017. KAFKA: The modern platform for data management and analysis in big data domain. In *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*. 1–5. https://doi.org/10.1109/TEL-NET.2017.8343593

[61] Petru Soviany, Radu Tudor Ionescu, Paolo Rota, and Nicu Sebe. 2022. Curriculum Learning: A Survey. arXiv:2101.10382 [cs.LG] https://arxiv.org/abs/2101.10382

[62] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2025. Gemini: a family of highly capable multimodal models. arXiv:2312.11805 [cs.CL] https://arxiv.org/abs/2312.11805

[63] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. 2023. FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline. *Proc. VLDB Endow.* 16, 5 (jan 2023), 1086–1099.

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[65] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. 2024. Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 195–210.

[66] Borui Wan, Mingji Han, Yiyao Sheng, Zhichao Lai, Mofan Zhang, Junda Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. 2024. ByteCheckpoint: A Unified Checkpointing System for LLM Development. arXiv:2407.20143 [cs.AI] https://arxiv.org/abs/2407.20143

[67] Borui Wan, Gaohong Liu, Zuquan Song, Jun Wang, Yun Zhang, Guangming Sheng, Shuguang Wang, Houmin Wei, Chenyuan Wang, Weiqiang Lou, Xi Yang, Mofan Zhang, Kaihua Jiang, Cheng Ren, Xiaoyun Zhi, Menghan Yu, Zhe Nan, Zhuolin Zheng, Baoquan Zhong, Qinlong Wang, Huan Yu, Jinxin Chi, Wang Zhang, Yuhan Li, Zixian Du, Sida Zhao, Yongqiang Zhang, Jingzhe Tang, Zherui Liu, Chuan Wu, Yanghua Peng, Haibin Lin, Wencong Xiao, Xin Liu, and Liang Xiang. 2025. Robust LLM Training Infrastructure at ByteDance. arXiv:2509.16293 [cs.LG] https://arxiv.org/abs/2509.16293

[68] Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. 2024. Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution. arXiv:2409.12191 [cs.CV] https://arxiv.org/abs/2409.12191

[69] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 364–381. https://doi.org/10.1145/3600006.3613145

[70] Chengyue Wu, Xiaokang Chen, Zhiyu Wu, Yiyang Ma, Xingchao Liu, Zizheng Pan, Wen Liu, Zhenda Xie, Xingkai Yu, Chong Ruan, and Ping Luo. 2024. Janus: Decoupling Visual Encoding for Unified Multimodal Understanding and Generation. arXiv:2410.13848 [cs.CV] https://arxiv.org/abs/2410.13848

[71] Zhiyu Wu, Xiaokang Chen, Zizheng Pan, Xingchao Liu, Wen Liu, Damai Dai, Huazuo Gao, Yiyang Ma, Chengyue Wu, Bingxuan Wang, Zhenda Xie, Yu Wu, Kai Hu, Jiawei Wang, Yaofeng Sun, Yukun Li, Yishi Piao, Kang Guan, Aixin Liu, Xin Xie, Yuxiang You, Kai Dong, Xingkai Yu, Haowei Zhang, Liang Zhao, Yisong Wang, and Chong Ruan. 2024. DeepSeek-VL2: Mixture-of-Experts Vision-Language Models for Advanced Multimodal Understanding. arXiv:2412.10302 [cs.CV] https://arxiv.org/abs/2412.10302

[72] Jiasheng Ye, Peiju Liu, Tianxiang Sun, Yunhua Zhou, Jun Zhan, and Xipeng Qiu. 2024. Data Mixing Laws: Optimizing Data Mixtures by Predicting Language Modeling Performance. arXiv:2403.16952 [cs.CL] https://arxiv.org/abs/2403.16952

[73] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (Oct. 2023), 148–161. https://doi.org/10.14778/3626292.3626298

[74] Zili Zhang, Yinmin Zhong, Yimin Jiang, Hanpeng Hu, Jianjian Sun, Zheng Ge, Yibo Zhu, Daxin Jiang, and Xin Jin. 2025. DistTrain: Addressing Model and Data Heterogeneity with Disaggregated Training for Multimodal Large Language Models. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 24–38.

[75] Mark Zhao, Emanuel Adamiak, and Christos Kozyrakis. 2024. cedar: Optimized and Unified Machine Learning Input Data Pipelines. *Proc. VLDB Endow.* 18, 2 (2024), 488–502.

[76] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 1042–1057.

[77] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (2023), 3848–3860.

[78] Yu Zhu, Wenqi Jiang, and Gustavo Alonso. 2025. Multi-Tenant SmartNICs for In-Network Preprocessing of Recommender Systems. arXiv:2501.12032 [cs.AR] https://arxiv.org/abs/2501.12032

[79] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini. 2024. Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 761–774. https://www.usenix.org/conference/nsdi24/presentation/zu