Blueprint for a Modern High-Performance Machine Learning System

I. Executive Summary: Blueprint for a Modern ML System

The design of a modern, large-scale Machine Learning (ML) system demands a holistic approach, where compute, networking, and storage are not merely individual components but deeply interconnected elements working in synergy. Achieving optimal performance, scalability, and efficiency for demanding ML workloads hinges on this integrated design philosophy. This report outlines a blueprint for such a system, emphasizing key technology choices that address the current and emerging needs of advanced AI applications. These choices include high-performance Ethernet with RDMA over Converged Ethernet (RoCEv2) or InfiniBand for the network fabric, a tiered storage architecture featuring high-speed parallel file systems for active training data and scalable object storage for vast datasets, powerful GPU clusters equipped with advanced interconnects like NVIDIA NVLink, and a robust MLOps framework to ensure reproducibility and operational efficiency.

The core challenge in contemporary ML systems extends beyond the raw power of individual components. While cutting-edge GPUs provide immense computational capability, their effective utilization is often constrained by the ability of the network to deliver data and the storage system to serve it at the required pace.[1] Scalability, the ability to grow the system to handle larger models and datasets; reproducibility, the capacity to consistently replicate experimental results and model behavior; and efficiency, optimizing resource use and minimizing training times, are paramount design goals.[3]

A critical understanding is that optimizing components in isolation

yields suboptimal system-level outcomes. For instance, a state-of-the-art GPU cluster will remain underutilized if the network bandwidth is insufficient or if the storage Input/Output Operations Per Second (IOPS) create a bottleneck.[5] This necessitates a design where budget allocation and engineering efforts are distributed across all critical subsystems—compute, network, and storage—rather than being disproportionately focused on a single area like GPU acquisition. Furthermore, the modern ML infrastructure is increasingly software-defined. Beyond the physical hardware, the selection of workload managers, MLOps tools, and data orchestration layers significantly shapes the system's capabilities, operational agility, and overall effectiveness.[6] Building and operating such sophisticated systems requires a multidisciplinary team with expertise spanning hardware engineering, distributed software systems, and MLOps practices.

II. Core Compute Infrastructure: GPU Clusters for Intensive ML

The heart of any large-scale ML system is its compute infrastructure, predominantly comprising clusters of Graphics Processing Units (GPUs) tailored for the parallel processing demands of ML algorithms, particularly deep learning.

A. GPU Selection and Server Configuration

The selection of GPUs is a foundational decision, directly impacting the system's performance, cost, and ability to handle specific ML workloads. Leading GPU architectures for ML include NVIDIA's H100, H200, and the newer Blackwell series, alongside AMD's MI300X. Key evaluation criteria include raw compute power (measured in TFLOPS for various precisions like FP16, BF16, FP8, and INT8), on-chip memory capacity (High Bandwidth Memory - HBM), memory

bandwidth, and power consumption (Thermal Design Power - TDP).[10]

For instance, the NVIDIA H100 GPU offers 80GB of HBM2e or HBM3 memory, utilizes PCIe Gen5 for host connectivity, and features configurable power management modes to balance performance and energy use.[10] Cloud providers like Google Cloud offer A3 virtual machines equipped with NVIDIA H100 80GB GPUs.[11] In comparison, the AMD MI300X boasts a significant 192GB of HBM3 memory, offering superior memory capacity and bandwidth over the H100 80GB, making it particularly well-suited for training very large models that might otherwise require complex model parallelism strategies.[12] Optimizing workloads on MI300X can involve specific configurations, such as adjusting gpu-memory-utilization, max-num-seqs for batching, and selecting appropriate data types (dtype) when using frameworks like vLLM.[13]

Server configurations must be carefully matched to the chosen GPUs. This includes selecting between PCIe and SXM form factors (with SXM typically offering higher GPU-to-GPU bandwidth within a multi-GPU baseboard), appropriate CPUs (e.g., Intel Sapphire Rapids, AMD Epyc Genoa [6]), sufficient system RAM, and robust power delivery and cooling systems within the server chassis to handle the high TDP of modern GPUs.[14]

The trend towards increasingly large models, especially Large Language Models (LLMs), places GPU memory capacity and bandwidth at the forefront of selection criteria.[6] GPUs with larger memory, like the AMD MI300X, can reduce the necessity for extensive model parallelism across numerous GPUs with smaller memory footprints. This simplification of distributed training setups can, in turn, lessen inter-node communication overhead and

potentially improve the total cost of ownership (TCO) by allowing complex models to fit onto fewer nodes or even a single, powerful node. This shifts the design focus from solely raw TFLOPS to TFLOPS accessible per model parameter, a metric intrinsically linked to available GPU memory.

While raw hardware specifications are crucial, the maturity and breadth of the supporting software ecosystem are equally important. NVIDIA's CUDA platform, along with its extensive suite of libraries (e.g., Magnum IO, NCCL, cuDNN, TensorRT [15]), provides a highly developed environment for ML development and deployment. AMD is actively building its ROCm ecosystem and collaborating with communities like vLLM to enhance support.[12] However, the established nature of NVIDIA's ecosystem often translates to a smoother development experience, wider framework compatibility, and more readily available pre-optimized solutions, which can accelerate time-to-solution and simplify troubleshooting in production environments. Thus, the GPU selection process involves a careful assessment of not just hardware capabilities but also the associated software stack and ecosystem support.

Table 1: GPU Model Comparison for ML Workloads

Feature	**NVIDIA H100 80GB (SXM5)**	**NVIDIA H200 141GB (SXM)**	**AMD MI300X 192GB**	**NVIDIA Blackwell B200 (Projected)**
Architecture	Hopper	Hopper	CDNA 3	Blackwell

FP16/BF16 TFLOPS	~1979 (dense)	~1979 (dense)	~1600-1900 +	Significantly Higher
FP8 TFLOPS	~3958 (dense)	~3958 (dense)	~3200-380 0+	Significantly Higher
HBM Capacity	80 GB HBM3	141 GB HBM3e	192 GB HBM3	Up to 192GB HBM3e
HBM Bandwidth	3.35 TB/s	4.8 TB/s	5.3 TB/s	Up to 8 TB/s
Interconnect	NVLink 4.0 (18 links)	NVLink 4.0 (18 links)	Infinity Fabric	NVLink 5.0 (18 links)
Link Bandwidth (Total)	900 GB/s	900 GB/s	896 GB/s (IFL)	1.8 TB/s
Power (TDP)	Up to 700W	Up to 1000W	750W	Up to 1200W
Form Factor	SXM5	SXM	OAM	SXM / Server Board

Note: Blackwell B200 specifications are based on publicly announced information and may be subject to change. AMD MI300X TFLOPS can vary based on specific configurations and sparsity.

B. Intra-Node and Inter-Node GPU Connectivity (NVLink, NVSwitch, and

alternatives)

For multi-GPU training, particularly distributed deep learning, high-speed, direct GPU-to-GPU communication is paramount to minimize latency and maximize bandwidth for operations like gradient synchronization and model parameter sharing.

NVIDIA's NVLink technology is a key enabler here. The fifth generation of NVLink, featured in the Blackwell architecture, offers up to 1.8 TB/s of bidirectional bandwidth per GPU, facilitated by 18 links each operating at 100 GB/s.[17] This represents a twofold increase over the previous generation and is more than 14 times the bandwidth of PCIe Gen5.[17] To scale this further, NVIDIA NVSwitch chips connect multiple NVLink interfaces, enabling all-to-all GPU communication at full NVLink speed, not only within a single server but also potentially between servers in a tightly coupled pod.[17] For example, the NVSwitch 3 chip integrates 64 NVLink4 ports.[18] NVIDIA has also productized this technology into physical NVLink Switches, which can connect multiple GPU servers to form an "NVLink Network," effectively creating a high-speed fabric dedicated to GPU communication across hosts.[18] Platforms like the NVIDIA GB200 or GB300 NVL72 systems, which can link up to 72 GPUs, heavily rely on this advanced NVLink and NVSwitch infrastructure to function as a cohesive, powerful compute unit.[17] AMD offers its Infinity Fabric and Infinity Links for inter-GPU communication within its own ecosystem, providing a competing solution for high-bandwidth connectivity, though detailed specifications for the latest generations were less prominent in the provided materials.

NVIDIA's substantial investment in NVLink and NVSwitch technology has resulted in a highly integrated, high-performance ecosystem for

multi-GPU servers and pods, such as their DGX, HGX, and the newer NVL72 systems.[17] This provides a potent "scale-up" solution, allowing for extremely fast communication within a pod of GPUs before needing to "scale-out" over a traditional network fabric like Ethernet or InfiniBand. For certain model parallelism strategies (e.g., tensor or pipeline parallelism), this localized high-bandwidth, low-latency interconnect can be particularly advantageous, simplifying aspects of distributed training by making inter-GPU communication within the pod exceptionally efficient. The NVLink Switch, for example, can enable 130TB/s of GPU bandwidth in a single GB300 NVL72 system, supporting up to nine times the GPU count of a typical 8-GPU system for large model parallelism, effectively creating a "data-center-sized GPU".[17]

This evolution is leading to the concept of "rack-scale GPUs." Technologies like the external NVLink Switch [18] are blurring the traditional boundaries between individual servers. They create larger, more cohesive computational units at the rack or multi-rack level, where up to 576 GPUs can be interconnected in a non-blocking fabric via NVLink.[17] This has significant implications for network design *within* the AI cluster. For these tightly coupled GPU pods, the primary high-performance interconnect is NVLink, handling the most intense, fine-grained communication. The external network fabric, such as Ethernet or InfiniBand, then serves to connect these powerful "rack-scale GPU" pods to each other and to the storage infrastructure. This implies a hierarchical networking approach, with specialized networks for different communication patterns and distances.

C. Cluster Management and Orchestration (Kubernetes, Slurm, Ray – selection criteria)

Managing and orchestrating workloads across a large GPU cluster requires sophisticated software. Key players in this domain include Kubernetes, Slurm, and Ray, each with distinct strengths.

- **Kubernetes:** Has become a de facto standard for container orchestration. It can manage GPU resources, scale applications, and offers features like node labeling for different GPU types and mechanisms for specifying GPU resource requests in pod definitions.[6] Google Kubernetes Engine (GKE), for example, provides managed Kubernetes with GPU support, including features like time-sharing and Multi-Instance GPUs (MIG) for partitioning physical GPUs into smaller, isolated instances.[19] However, for specialized GenAI workloads involving complex job queuing and scheduling, Kubernetes often requires supplementary tools or operators.[6]
- **Slurm (Simple Linux Utility for Resource Management):** A widely adopted workload manager in High-Performance Computing (HPC) environments, Slurm excels at automating task scheduling, managing batch processing queues, and optimizing cluster resource utilization.[6] It manages GPUs as Generic Resources (GRES) and utilizes environment variables like CUDA_VISIBLE_DEVICES and CUDA_DEVICE_ORDER for GPU allocation and visibility.[21]
- **Ray:** An open-source framework designed for scaling Python applications, particularly those in AI and ML.[6] Ray provides a suite of libraries tailored for ML tasks, including Ray Train for distributed model training, Ray Tune for hyperparameter optimization, Ray Data for scalable data processing, and Ray Serve for model deployment.[6] Ray can be deployed on various infrastructures, including on top of Spark clusters (as seen with

Databricks [23]) or in a more infrastructure-agnostic manner using Docker containers.[22]

The selection of an orchestrator depends on several factors: the existing expertise within the team, the nature of the ML workloads (e.g., batch training, interactive development, model serving), integration with existing infrastructure, the desired level of abstraction from the underlying hardware, and compatibility with other MLOps tools.

A notable trend is the convergence and specialization of these orchestration tools. While Kubernetes provides a robust foundational layer for container and resource management, specialized frameworks like Slurm (for HPC-style batch jobs) and Ray (for distributed Python/ML workloads) offer functionalities more finely tuned to specific ML tasks. Consequently, integrations such as Soperator (a Kubernetes operator for Slurm, enabling Slurm clusters to run on Kubernetes [6]) or deploying Ray clusters on Kubernetes are becoming common. This layered approach allows organizations to leverage the general-purpose infrastructure management capabilities of Kubernetes while benefiting from the domain-specific scheduling features and rich ML ecosystems offered by tools like Slurm or Ray.

The choice of orchestrator also has a direct bearing on the integration and workflow of the broader MLOps toolkit. For instance, Kubeflow, with its components like Kubeflow Pipelines, is inherently Kubernetes-native and designed for orchestrating ML workflows on Kubernetes.[24] MLflow, on the other hand, offers greater flexibility, capable of integrating with various backend stores and artifact repositories, and can be used with different orchestrators.[8] Ray

comes with its own ecosystem of tools like Ray Tune and Ray Serve, which are naturally integrated within Ray applications.[22] Therefore, the decision regarding the primary orchestrator is not merely about job execution but significantly shapes the entire MLOps landscape, influencing how experiments are tracked, data and models are versioned, and models are ultimately deployed.

III. High-Performance Networking Fabric: The Data Superhighway

The network fabric is the critical data superhighway connecting GPU clusters, storage systems, and preprocessing nodes. For large-scale ML, particularly distributed training, the network's characteristics directly impact overall system performance and efficiency.

A. Defining Network Requirements for Large-Scale ML (Bandwidth, Latency, Losslessness, Jitter)

Modern AI applications, especially those involving distributed training of large models, impose stringent requirements on the network.[5] Key performance indicators include:

- **Bandwidth:** The data transfer rate, typically ranging from 100Gbps to 400Gbps, and evolving towards 800Gbps and beyond per link.[5] High bandwidth is essential for moving large datasets from storage to compute nodes and for synchronizing large model parameters and gradients between GPUs during distributed training.
- **Latency:** The delay in data transmission. Extremely low latency is crucial for synchronous operations in distributed training, such as the aggregation of gradients, where delays can cause GPUs to idle, significantly slowing down the training process. Technologies like RoCE v2 aim to reduce latency by enabling direct memory access and bypassing the operating system and

CPU.[27] InfiniBand is also renowned for its inherently low latency characteristics.[28]

- **Losslessness:** The absence of packet loss during transmission. Packet loss leads to retransmissions, which can stall training computations and severely degrade performance. Lossless operation is typically achieved in Ethernet networks through mechanisms like Ethernet Flow Control or, more commonly for RoCE, Priority Flow Control (PFC) combined with Explicit Congestion Notification (ECN).[29] Arista's EOS, for example, provides tools to achieve a highly reliable, lossless network.[5] InfiniBand, by contrast, employs a credit-based flow control mechanism that inherently ensures lossless communication between connected devices.[29]
- **Jitter:** The variation in packet arrival times. Consistent and predictable packet delivery (low jitter) is important for maintaining stable performance in tightly synchronized distributed computations. The Ultra Ethernet Consortium (UEC) identifies low jitter as a key design goal for future AI networks.[30]

Achieving a truly "lossless" network, especially for RoCEv2 deployments, is a system-level challenge, not merely a feature of the switch hardware. It necessitates meticulous end-to-end configuration and tuning, encompassing NICs, switches, and sophisticated congestion control mechanisms.[5] This complexity is a trade-off for leveraging the ubiquity and broader ecosystem of Ethernet. InfiniBand's credit-based flow control, in contrast, offers a more inherently lossless fabric.

Furthermore, the role of the network is evolving beyond that of a passive data conduit. Emerging technologies like NVIDIA SHARP (Scalable Hierarchical Aggregation and Reduction Protocol), which

can be integrated into NVSwitch chips [18] and is part of the Magnum IO software stack [16], enable "in-network computing." With SHARP, collective operations like reductions (summing gradients from multiple GPUs) can be performed within the network fabric itself as data transits. This offloads computation from the GPUs or CPUs, potentially significantly accelerating communication patterns common in distributed training. This trend positions the network as an active participant in the computation, making the switch more than just a data forwarder—it becomes an accelerator.

B. Key Networking Technologies: Ethernet with RoCE v2, InfiniBand, and the emerging Ultra Ethernet Consortium (UEC)

Several key technologies compete to provide the networking backbone for ML clusters:

- **Ethernet with RoCE v2 (RDMA over Converged Ethernet):** This approach leverages standard Ethernet infrastructure, which is widely deployed and understood. RoCE v2 is a routable protocol, typically operating over UDP/IP on port 4791.[29] It enables Remote Direct Memory Access (RDMA), allowing systems (e.g., storage servers and GPU nodes, or GPU nodes amongst themselves) to exchange data directly between their memories, bypassing the host CPU and operating system networking stack. This significantly reduces latency and CPU overhead.[27] For example, Google Cloud's A3 and A4 VMs utilize RoCE v2 to achieve 1.6 Tbps to 3.2 Tbps of inter-node GPU-to-GPU traffic.[27] Network vendors like Arista provide switches (e.g., their Etherlink portfolio) that support RoCE and the necessary features for building lossless Ethernet fabrics, such as PFC and ECN.[5]
- **InfiniBand:** A high-throughput, very low-latency interconnect

technology that has long been favored in HPC environments and has seen widespread adoption in large-scale AI clusters.[28] InfiniBand inherently supports RDMA and uses a credit-based flow control mechanism, which guarantees lossless communication without the complex configuration often required for lossless Ethernet.[29] NVIDIA's Quantum-2 InfiniBand platform, for instance, offers switches with aggregate throughputs of 51.2 Tb/s and 400 Gb/s ports.[32] Historically, InfiniBand switches have demonstrated lower port-to-port latencies compared to their Ethernet counterparts.[29] As of early 2024, a significant majority, around 90%, of AI system deployments were reported to be using InfiniBand.[32]

- **Ultra Ethernet Consortium (UEC):** This is an industry initiative aimed at evolving and enhancing Ethernet specifically for the demands of AI and ML workloads.[30] The UEC's goal is to deliver an Ethernet-based solution with improved characteristics in terms of low latency, high bandwidth, minimal jitter, and enhanced reliability. The proposed UEC specifications cover multiple layers, including a physical layer compatible with IEEE 802.3 Ethernet, a link layer introducing Link Level Retry (LLR) for lossless transmission (potentially without relying on PFC), Packet Rate Improvement (PRI) through header compression, and an advanced transport layer featuring new congestion control mechanisms and support for out-of-order delivery.[33] Companies like Arista are actively involved and are building products that will be compatible with UEC standards.[5]

The choice between these technologies involves navigating a dynamic landscape. InfiniBand currently holds a strong position in AI clusters due to its mature RDMA implementation, inherently low

latency, and proven lossless nature.[32] However, Ethernet, augmented by RoCEv2 and the forthcoming UEC enhancements, is rapidly evolving to challenge this dominance. Ethernet offers the allure of leveraging a ubiquitous, well-understood technology base, potentially broader vendor choice, and potentially lower costs in some scenarios.[5] Hyperscalers, for example, often prefer Ethernet due to its openness and ability to handle diverse workloads beyond just AI.[32] NVIDIA itself is a proponent of "lossless Ethernet" through its Spectrum-X platform.[32] The decision for a new ML cluster build becomes strategic: investing in the established performance and relative simplicity of InfiniBand or opting for the evolving Ethernet/UEC path, which promises future advancements within a more common data center fabric.

Regardless of the specific fabric choice (InfiniBand or Ethernet/RoCEv2), RDMA capability is a fundamental and non-negotiable requirement for any high-performance ML network.[27] The ability to bypass CPU and OS overhead for data transfers between GPUs and between GPUs and storage is critical for achieving the efficiency and speed demanded by large-scale distributed training.

Table 2: Networking Technology Comparison for ML Clusters

Feature	Ethernet + RoCEv2	InfiniBand	Ultra Ethernet (Projected)
Typical Bandwidth/Port	100G, 200G, 400G, 800G+	100G (HDR100), 200G (HDR),	400G, 800G, 1.6T+

		400G (NDR), 800G (XDR)	
Latency Characteristics	Low (with RDMA), higher than InfiniBand typically	Very Low	Very Low (target similar/better than InfiniBand)
Lossless Mechanism	PFC, ECN, DCB/DCB-CX (complex configuration) [29]	Credit-based flow control (inherent) [29]	UEC Transport (e.g., LLR) [33]
RDMA Support	Yes (RoCEv2) [27]	Yes (Native) [28]	Yes (Enhanced Ethernet RDMA)
Ecosystem Maturity	Very Mature (Ethernet), Growing (RoCE for AI)	Mature (HPC & AI)	Emerging
Cost Considerations	Potentially lower, wider vendor base	Typically higher cost per port	Aims for Ethernet cost-effectiveness
Key AI/ML Use Cases	General DC, AI/ML clusters, Hyperscalers [32]	Dedicated AI/ML clusters, HPC [32]	Future AI/ML clusters

C. Network Topologies for Scalability and Performance (e.g., Fat-Tree, Clos, Dragonfly)

The physical and logical arrangement of network switches and links—the network topology—is crucial for ensuring scalability, high performance, and fault tolerance in large GPU clusters.

- **Fat-Tree (and its common implementation, Spine-Leaf, a type of Clos network):** This is the most commonly adopted topology for modern data centers and GPU clusters due to its excellent scalability, potential for non-blocking or low-blocking performance, and relatively simple routing.[34]
    - A **two-tier Fat-Tree (Leaf-Spine)** architecture is typically used for smaller to medium-sized clusters. In this setup, servers (leaf nodes) connect to a set of leaf switches. Each leaf switch then connects to every spine switch in the upper tier. This provides multiple paths and good aggregate bandwidth.[34]
    - For larger clusters, a **three-tier Fat-Tree (Leaf-Spine-Core)** architecture may be employed, adding another layer of core switches to interconnect multiple spine-leaf pods.[34]
    - The number of GPUs a Fat-Tree network can support in a non-blocking fashion is directly related to the port count (radix, P) of the switches used. For a two-tier non-blocking network where leaf switches use P/2 ports downwards to servers and P/2 ports upwards to spine switches, a maximum of P leaf switches can be supported, leading to a total of $P \times (P/2) = P^2/2$ server ports (assuming one port per server/GPU for simplicity).[34] For example, using 40-port switches, a two-tier Fat-Tree can support up to $40^2/2 = 800$ GPU connections. With 128-port switches, this scales to $128^2/2 = 8192$ GPUs.[34] A three-tier non-blocking Fat-Tree can

support up to P3/4 server ports.[34]
    - The **Clos network**, named after Charles Clos, is the general theoretical underpinning for such multi-stage switching networks.[37] A key property is that a strictly non-blocking Clos network can be achieved if the number of middle-stage switches (m) is greater than or equal to 2n−1, where n is the number of inputs to each ingress switch.[37] Spine-leaf architectures are practical implementations of Clos principles, designed to minimize the number of hops between any two endpoints, thus ensuring high-bandwidth and low-latency communication ideal for data centers and AI clusters.[36]
- **Dragonfly:** This topology, popularized by Cray in their supercomputers (e.g., XC and EX series), employs a hierarchical structure of switch groups.[38] Within each group (often corresponding to a cabinet or set of cabinets), switches are typically connected in an all-to-all manner. These groups are then interconnected, often with a reduced (tapered) number of links compared to the intra-group connectivity, forming a global network.[38] Dragonfly topologies are designed to offer very high bisection bandwidth, which is a measure of the network's capacity for all-to-all communication, and can provide lower average latency compared to Fat-Trees for certain communication patterns, especially in very large systems.[39] They are particularly effective for physically dense node configurations, as more connections can be made with shorter, less expensive cables within a group.[38] However, Dragonfly networks can be more complex to design, cable, and manage than Fat-Trees, and their benefits might diminish if the physical density of nodes is low, requiring more optical connections.[38]

The choice of topology involves a trade-off. For many enterprise and research AI clusters, a well-designed Fat-Tree/Clos network (typically spine-leaf) provides a good balance of predictable performance, scalability, and manageability.[34] For extremely large-scale systems, such as those found in national supercomputing centers or at hyperscalers with workloads dominated by intense all-to-all communication, a Dragonfly topology might be considered for its potential advantages in bisection bandwidth, though at the cost of increased complexity.[38]

A critical factor influencing the scale and cost-effectiveness of any chosen topology is the **radix** (number of ports) of the network switches. Higher-radix switches allow for the construction of larger non-blocking or low-blocking fabrics with fewer switching layers.[34] For example, as shown by the formulas for Fat-Tree capacity, a switch with 64 ports can support significantly more endpoints in a two-tier non-blocking fabric than a 32-port switch, potentially obviating the need for a more complex and costly three-tier topology for a given cluster size. The availability of very high-radix spine switches, like the Arista 7800R4 AI Spine with 576 ports of 800G [5], enables the creation of very large, efficient single-switch clusters (for smaller deployments) or forms the core of massive multi-tier networks. Investing in higher-radix switches can, therefore, simplify network design, reduce the number of hops, decrease overall latency, and potentially lower the total cost of the network infrastructure.

IV. Scalable and Tiered Storage Architecture for ML Data Lifecycle

Machine learning systems generate and consume vast quantities of

data throughout their lifecycle, from raw data ingestion to model training, checkpointing, and artifact archiving. An effective storage architecture must balance performance, capacity, and cost, typically through a tiered approach.[40] Key characteristics for ML storage include scalability to handle growing data volumes, high availability to ensure uninterrupted access, robust security mechanisms, and performance tailored to different stages of the ML pipeline, encompassing both high throughput for large sequential reads/writes and low latency for metadata operations and small random accesses.[1]

A. Overall Storage Strategy: Tiers (Hot, Warm, Cold) and Data Placement

A multi-tiered storage strategy is essential for managing the diverse requirements of ML workloads:

- **Hot Tier:** This tier houses data requiring the highest performance, such as active training datasets, frequently accessed model checkpoints, and temporary scratch space. Technologies typically include local NVMe SSDs on compute nodes or high-performance parallel file systems. The primary focus is on minimizing latency and maximizing throughput to keep GPUs fed with data.[1]
- **Warm Tier:** This tier provides scalable storage for data that is accessed less frequently but still needs to be readily available. Examples include larger, less active portions of training datasets, archived model artifacts, and historical experiment data. Solutions might include general-purpose distributed file systems or object storage systems, often augmented with caching layers to improve access times. This tier balances performance with cost-effectiveness.
- **Cold Tier:** This is the most cost-effective tier, designed for

long-term archival of raw data, older model versions, logs, and other data that is rarely accessed. Cloud-based archival services like Amazon S3 Glacier or Azure Archive Storage are common choices.

Data placement across these tiers should ideally be dynamic and automated. As dataset sizes explode into petabytes [6], manual management becomes infeasible. Intelligent tiering, driven by policies based on access frequency, last modification time, data size, or even predictive models using machine learning itself, is crucial for ensuring that data resides on the most appropriate storage class.[40] This optimizes both performance (by keeping active data on fast tiers) and cost (by moving inactive data to cheaper tiers). Such automated tiering is becoming a key feature of modern storage solutions designed for large-scale data environments.

The storage architecture must support the entire data lifecycle seamlessly. This includes the initial ingestion of raw data, preprocessing and transformation stages, the iterative process of model training with frequent checkpointing, storage of the final trained model artifacts, and eventual archival of data and models. Each of these stages imposes different I/O patterns and performance demands on the storage system.[1] For example, raw data might initially land on scalable object storage [1], preprocessing might leverage a distributed file system or local scratch space [42], training requires high, sustained throughput from a parallel file system [44], model checkpoints need fast write capabilities [46], and model artifacts require versioned and easily accessible storage.[8] This implies that the storage architecture is not a monolithic entity but rather a collection of specialized systems managed cohesively to serve the end-to-end ML workflow.

B. High-Performance File Systems for Training Data (e.g., Lustre, BeeGFS, IBM Spectrum Scale)

For the hot tier, particularly for feeding large datasets to GPU clusters during training, parallel file systems are indispensable. These systems are architected to provide high-throughput, low-latency, and highly concurrent access to massive datasets, distributing data and I/O operations across multiple storage nodes and servers.[1]

- **Lustre:** A widely deployed open-source parallel file system, particularly prevalent in HPC and increasingly in large-scale LLM training environments. Lustre can scale to petabytes of capacity and deliver aggregate throughput in the terabytes per second range.[48] Managed cloud offerings, such as Google Cloud Managed Lustre (based on DDN EXAScaler technology), provide performance tiers like 1000 MB/s per Terabyte (TiB) of provisioned capacity.[44] Amazon FSx for Lustre is another popular managed service, which notably supports integration with Elastic Fabric Adapter (EFA) and NVIDIA GPUDirect Storage (GDS). This integration allows FSx for Lustre to deliver up to 1200 Gbps of throughput per client instance to compatible EC2 GPU instances, significantly accelerating data access for training.[49]
- **BeeGFS:** An open-source parallel file system known for its ease of deployment and linear scalability in both performance and capacity.[51] It is well-suited for ML and AI workloads that demand high data throughput. A notable deployment at UCSB demonstrated over 13 GB/s of performance.[51] BeeGFS also offers a feature called BeeOND (BeeGFS On Demand), which allows for the creation of temporary, high-performance file system

instances on the local SSDs of compute nodes, serving as a burst buffer for I/O-intensive jobs.[51]

- **IBM Spectrum Scale (formerly GPFS):** A robust, software-defined storage solution providing both file and object access through its massively parallel file system.[45] Spectrum Scale is designed for high-performance workloads and supports NVIDIA GPUDirect Storage, enabling a direct data path between GPU memory and storage over InfiniBand or RoCE networks. This direct path allows data to be read from or written directly to an NSD (Network Shared Disk) server's pagepool and transferred to the GPU buffer of client nodes via RDMA, bypassing the CPU.[45] It is used in demanding environments like the IBM Vela AI supercomputer and can scale to thousands of nodes.[45]

A critical feature for modern parallel file systems serving ML training workloads is their integration with NVIDIA GPUDirect Storage (GDS).[56] The ability of Lustre [49], IBM Spectrum Scale [54], and other parallel file systems [58] to facilitate direct data transfers to GPU memory significantly reduces CPU overhead and data access latency, which is becoming a standard requirement for achieving optimal training performance. This capability heavily influences the selection of a parallel file system, placing pressure on vendors to offer robust GDS implementations.

The choice of a specific parallel file system can also be influenced by factors such as cloud provider offerings and existing institutional expertise. Managed services like Google Cloud Managed Lustre [44] or AWS FSx for Lustre [49] significantly lower the operational burden of deploying and maintaining these complex systems, making them attractive for cloud-native ML initiatives. Conversely, organizations

with established HPC infrastructure may already possess deep expertise in Lustre, BeeGFS, or Spectrum Scale, guiding their choices for new ML cluster deployments based on existing skill sets, vendor relationships, and specific performance or feature requirements. Thus, the "best" parallel file system is context-dependent.

Table 3: Parallel File System Comparison for ML Training

Feature	Lustre	BeeGFS	IBM Spectrum Scale (GPFS)
Architecture Highlights	Distributed metadata (MDS) & object storage (OSS) servers [48]	Distributed metadata & storage servers; client-side striping [51]	Shared-disk cluster file system; NSDs; File & Object access [45]
Typical Throughput	TBs/sec (aggregate) [48]; 1000 MB/s/TiB (managed services) [44]	Tens of GBs/sec to TBs/sec (aggregate) [51]	TBs/sec (aggregate with multiple nodes) [53]
Latency Characteristics	Low for large I/O; metadata can be a bottleneck if not scaled	Low latency, good for mixed I/O	Low latency, optimized for parallel I/O

Scalability (Capacity/Nodes)	Petabytes; thousands of clients [48]	Petabytes; thousands of nodes [51]	Exabytes; thousands of nodes [53]
GPUDirect Storage Support	Yes (e.g., AWS FSx for Lustre with EFA/GDS [49])	Yes (via GDS-compatible underlying block devices & client integration)	Yes (Direct RDMA from NSD server pagepool to GPU buffer) [54]
Key ML/HPC Use Cases	LLM Training, HPC simulations, Genomics [48]	AI/ML, Deep Learning, Lifesciences, HPC [51]	AI, HPC, Analytics, Global Data Platforms [45]
Management Model	Open Source; Managed Cloud Services (AWS, GCP, Oracle) [44]	Open Source; Commercial Support Available [51]	Commercial Software; Appliance options (Storage Scale System) [53]

C. Object Storage for Raw Datasets, Archives, and Large Artifacts (e.g., S3-compatible with caching layers like Alluxio/JuiceFS)

Object storage systems, such as Amazon S3, Azure Blob Storage, Google Cloud Storage, and S3-compatible on-premises solutions like MinIO, have become the cornerstone for storing vast quantities of unstructured data in ML environments.[1] Their inherent scalability (virtually unlimited capacity), durability, and cost-effectiveness make them ideal for building data lakes that house raw training datasets, intermediate processed data, large model artifacts, and long-term archives.[60] The S3 API has emerged as the de-facto industry

standard for object storage interaction.[60]

While object storage excels as a primary data lake foundation due to its scalability and cost profile [61], its native performance characteristics (particularly latency and sometimes throughput for highly concurrent access patterns) are often insufficient for direct, high-performance feeding of GPUs during model training.[62] This performance gap has led to the widespread adoption of caching and data orchestration layers that sit between the object store and the compute clusters. These layers bring data closer to the compute resources, accelerate access, and can provide more familiar file system semantics.

- **Alluxio:** An open-source data orchestration platform that functions as a distributed caching layer. It intelligently manages data across various storage tiers, including memory, SSDs, and HDDs, within the compute cluster or on adjacent nodes.[62] Alluxio can connect to underlying persistent storage systems like Amazon S3, GCS, or HDFS. For ML workloads using frameworks like Spark or Ray, Alluxio caches frequently accessed data, significantly improving I/O throughput and reducing the need for repeated access to remote object stores.[63] Its newer DORA (Decentralized Object Repository Architecture) utilizes consistent hashing for distributing both cached data and metadata lookups across worker nodes, enhancing scalability and eliminating single points of failure.[65] Alluxio can expose cached data via a FUSE interface, presenting it as a local folder to applications.[65]
- **JuiceFS:** A cloud-native distributed file system that leverages object storage (like S3) as its backend for data persistence and employs a separate, pluggable metadata engine (options include

Redis, TiKV, or a proprietary distributed engine for the Enterprise Edition).[66] JuiceFS provides full POSIX compatibility, allowing applications to interact with it as if it were a local file system. It features multi-level caching (local client cache, distributed cache in Enterprise Edition) to accelerate data access and can manage hundreds of billions of files within a single namespace, making it suitable for very large AI datasets.[66]

The provision of POSIX-compliant file system interfaces by these caching layers (JuiceFS natively, Alluxio via FUSE) is a crucial feature.[65] Many existing ML tools, libraries, and even deep learning frameworks are built with the expectation of interacting with data through standard file system APIs. These layers abstract the underlying object storage, making it transparent to the applications and significantly reducing the engineering effort required to adapt ML workloads to use object storage as their primary data source. This creates a practical two-tier active data strategy: the object store serves as the durable and scalable "source of truth," while a faster caching layer provides high-performance access for active training and data processing workloads.

D. Specialized Storage

Beyond broad categories like parallel file systems and object storage, specific use cases within an ML system often benefit from specialized storage solutions.

1. NFS for Model Artifacts and Checkpoints (with performance considerations)

Network File System (NFS) is a mature and widely used protocol for sharing files across a network. In ML systems, it often serves as a

convenient solution for storing model artifacts (trained model files, configurations), experiment outputs, and shared home directories for users.[42] MLflow, for example, supports NFS as a backend for its artifact store.[8]

While standard NFS implementations can be straightforward to set up, they can become a performance bottleneck if not adequately provisioned or optimized, especially when handling frequent writes of large checkpoint files or concurrent access from multiple nodes. To mitigate this, high-performance NFS solutions or specific configuration tuning is necessary. Optimizations can include using appropriate mount options (e.g., sync vs. async, though async carries risks of data loss on server crash), increasing the number of nfsd server threads, and leveraging NFS over RDMA (Remote Direct Memory Access) to reduce latency and CPU overhead on both the client and server.[46] Commercial platforms like Silk are designed to provide high-performance NFS (and SMB) storage specifically for demanding AI training workloads, including optimized checkpointing capabilities, often in cloud environments like Azure.[47]

The role of NFS in modern ML systems is thus shifting. While its ubiquity and ease of use make it suitable for MLOps tooling (like MLflow artifact stores) and general file sharing, it is generally not the preferred choice for the primary, high-throughput data path during active model training, where parallel file systems offer superior performance. However, for scenarios where extreme parallelism isn't the dominant requirement—such as storing model files that are read once at the start of an inference job or infrequently during development, or for managing smaller checkpoint files—a well-configured NFS server can be a practical solution.

2. Local NVMe for Scratch Space and Temporary Data (e.g., direct use, ZFS LocalPV, RAM disks)

Local Non-Volatile Memory Express (NVMe) SSDs, present on the compute nodes themselves, offer an extremely low-latency and high-throughput storage tier.[42] This makes them ideal for use as scratch space for intermediate data generated during complex preprocessing or training computations, for burst buffering (where data is temporarily staged on local NVMe before being written to slower shared storage, as with BeeGFS On Demand [51]), or for local caching of frequently accessed portions of a larger dataset.

Managing local NVMe resources, especially in containerized environments like Kubernetes, requires specific approaches:

- **Direct host path mounts:** Simplest method, but lacks Kubernetes PV integration and isolation.[43]
- **ZFS LocalPV:** In Kubernetes environments, ZFS LocalPV can be used to aggregate multiple local NVMe disks on a node into a unified ZFS pool. This pool can then be used for dynamic provisioning of persistent volumes (PVs) with ZFS's advanced features like data integrity and snapshotting, providing high-performance, node-local storage for pods.[43]
- **BeeGFS On Demand (BeeOND):** As mentioned, this BeeGFS feature can create temporary parallel file system instances using the local SSDs of compute nodes, offering a high-performance scratch space on a per-job basis.[51]

For scenarios demanding the absolute highest I/O performance for very temporary data, and where sufficient system RAM is available, **RAM disks** can be employed.[69] A RAM disk uses a portion of the system's main memory to emulate a block device. Since RAM is

orders of magnitude faster than even NVMe SSDs, this provides unparalleled speed for transient files. However, data stored on a RAM disk is volatile and will be lost upon system reboot or power loss.[70] Using RAM disks for temporary files can also help extend the lifespan of SSDs by reducing the number of write cycles they endure.[70]

Leveraging fast local NVMe for transient data, shuffle operations, or as a cache for frequently accessed data segments is a critical component of a tiered storage strategy. It complements slower, shared storage systems by offloading I/O-intensive operations that benefit from the lowest possible latency, thereby accelerating specific stages of ML pipelines.

3. Metadata Storage (e.g., XFS, ext4 for MLOps tools)

MLOps tools and other system components generate and manage significant amounts of metadata. This can include experiment parameters, metrics, run information, artifact locations tracked by tools like MLflow (if using a file-based backend store [26]), or the small .dvc files used by Data Version Control. The choice of file system for storing this metadata can impact the performance and reliability of these tools.

- **XFS:** A high-performance journaling file system designed for scalability and handling large files and parallel I/O operations efficiently. It can scale to exabytes of data and is well-suited for servers that store large individual files or need to handle many simultaneous I/O requests.[72] Its journaling capabilities ensure data integrity in case of system crashes.
- **Ext4 (Fourth Extended Filesystem):** A widely used, stable, and general-purpose journaling file system for Linux. It performs well

with smaller files and offers robust security features, including support for extended attributes and access control lists.[72]

The choice between XFS and ext4 for metadata storage depends on the specific I/O patterns of the MLOps tools and the nature of the metadata itself. If the metadata consists of many small files with frequent updates, ext4 might offer better performance for those specific operations. If the MLOps system involves storing larger log files or other substantial metadata artifacts, XFS's strengths in handling large files might be more beneficial. However, it's important to note that for scalable MLOps deployments, particularly with tools like MLflow, database-backed stores (e.g., PostgreSQL, MySQL) are generally recommended over file-based stores for metadata management due to better concurrency, queryability, and overall performance at scale.[71] The consideration of XFS vs. ext4 is more pertinent if a file-based backend is deliberately chosen, perhaps for simplicity in smaller or development setups.

V. Data Ingestion and Preprocessing Pipelines at Scale

The journey of data into an ML system, from raw sources through preprocessing to readiness for model training, is a critical phase that significantly influences overall pipeline efficiency and model quality.

A. Ingesting Diverse Data Sources (e.g., Apache Spark, Kafka for streaming)

Modern ML systems often need to ingest data from a multitude of sources, which can include databases, data warehouses, streaming platforms, APIs, and flat files. Efficient and scalable ingestion mechanisms are therefore essential.

- **Apache Spark:** A powerful open-source distributed processing system widely used for big data workloads, including Extract,

Transform, Load (ETL) operations.[74] Spark's core engine provides distributed task scheduling and execution. Spark SQL allows for querying structured and semi-structured data using SQL-like syntax, while Spark Streaming (and its successor, Structured Streaming) enables the processing of real-time data streams.[74] Spark's ability to connect to diverse data sources and perform complex transformations in parallel makes it a common choice for the initial stages of data ingestion and preparation in ML pipelines.

- **Apache Kafka:** A distributed event streaming platform designed for high-throughput, fault-tolerant, and scalable real-time data ingestion.[77] Kafka acts as a durable message queue, allowing various source systems to publish data streams (events or messages) to topics. Downstream applications, such as Spark jobs or other stream processors, can then subscribe to these topics and consume the data at their own pace. Kafka is often deployed as a central data ingestion layer, decoupling data producers from data consumers and providing a resilient buffer for incoming data.[77] Kafka Connect, a component of Kafka, provides a framework for scalably and reliably streaming data between Kafka and other systems like databases or file systems.[77]

Data ingestion pipelines typically involve more than just moving data; they often include initial data cleaning, validation, transformation, and formatting to prepare the data for subsequent preprocessing and model training stages.[78]

A robust architectural pattern involves decoupling the data ingestion mechanism from the subsequent processing and training systems. Using a dedicated ingestion layer like Apache Kafka provides such

decoupling.[77] Source systems can continuously publish data to Kafka topics, and downstream systems like Spark or Ray clusters can consume this data as needed. This architecture offers several advantages: it provides a buffer, absorbing bursts in data production; it allows data producers and consumers to evolve independently; and it enables multiple consumer applications to process the same data streams for different purposes. This loose coupling enhances the resilience and scalability of the overall data pipeline.

Furthermore, the rise of Generative AI, particularly models employing Retrieval Augmented Generation (RAG), introduces new complexities to the data ingestion phase.[79] For RAG systems, ingestion involves not only acquiring source documents but also chunking them into manageable pieces, generating vector embeddings for these chunks (often using another ML model), and storing these embeddings along with their metadata in a specialized vector store. This process, which includes NLP preprocessing and ML inference as part of the ingestion flow itself, represents an extension of traditional DataOps and requires specialized data pipelines.[79]

B. Distributed Data Preprocessing Frameworks (e.g., Ray Data, Dask)

Once raw data is ingested, it typically undergoes extensive preprocessing to transform it into a format suitable for ML model training. This can include operations like feature scaling, encoding categorical variables, handling missing values, text tokenization, image augmentation, and feature engineering. As dataset sizes grow, these preprocessing steps can become computationally intensive and, if not handled efficiently, can create a "CPU wall," bottlenecking the entire pipeline and leaving expensive GPUs idle during training.[80]

Distributed data preprocessing frameworks are designed to address this challenge by scaling out these CPU-bound tasks across multiple nodes.

- **Ray Data:** A scalable library within the Ray ecosystem specifically designed for data loading and preprocessing in ML workloads.[81] Ray Data supports streaming execution, which is beneficial for handling datasets larger than available memory and for minimizing latency between preprocessing and training. It integrates seamlessly with popular ML training frameworks like PyTorch, TensorFlow, and Hugging Face Transformers, allowing preprocessed data to be fed directly into training loops.[81] Ray Data can scale to petabyte-sized datasets and supports a wide variety of data transformations and input/output file formats, including Parquet, Lance, images, JSON, and CSV.[81] It can also be used in conjunction with caching layers like Alluxio to further accelerate data access from remote storage.[63] Several organizations, including Pinterest, DoorDash, and Instacart, utilize Ray Data for their ML data pipelines.[81]
- **Dask:** A flexible parallel computing library for Python that enables scalable analytics. Dask provides Dask DataFrames and Dask Arrays, which are parallel collections that mimic the APIs of Pandas DataFrames and NumPy arrays, respectively, allowing users to work with datasets larger than memory.[83] The dask-ml library includes scikit-learn-style transformers that can operate on these Dask collections in parallel, performing tasks like categorization (e.g., Categorizer for converting columns to categorical dtype) and encoding (e.g., DummyEncoder for one-hot encoding).[83] Dask can be integrated into workflow management tools like Luigi for building end-to-end data

pipelines [84] and can also run on Ray clusters via the dask_on_ray scheduler.[85]

- **Apache Spark:** As discussed in the ingestion section, Spark is also a powerful framework for large-scale data preprocessing.[74] Its MLlib library provides a suite of ML algorithms and utilities, including various tools for feature extraction, transformation, and selection, all designed to operate in a distributed manner.[74]

The "CPU wall" is a common scenario where the speed of data preprocessing on CPUs cannot keep pace with the consumption rate of fast GPUs, leading to GPU underutilization.[75] Distributed CPU-based preprocessing frameworks like Ray Data, Dask, and Spark are therefore essential components of a modern ML system, ensuring that data can be prepared and delivered to the training workers at a rate that matches the GPUs' processing capabilities.

Moreover, the emphasis on streaming execution, particularly in frameworks like Ray Data [81], is becoming increasingly important for handling extremely large datasets. Traditional batch ETL processes often involve processing entire datasets at each stage, which can be inefficient and require significant intermediate storage. Streaming execution, by contrast, allows for the overlapping of preprocessing and training stages: batches of data are preprocessed and immediately passed to the GPU workers. This reduces end-to-end latency, minimizes the need for storing massive intermediate datasets, and helps maintain high GPU utilization, especially when dealing with datasets that may not even fit into the distributed memory of the CPU cluster.

VI. Optimizing Data Flow to GPUs: Leveraging GPUDirect Storage

Minimizing the time it takes to move data from storage into GPU

memory is crucial for maximizing the utilization of expensive GPU resources and accelerating ML training. NVIDIA GPUDirect Storage (GDS) is a key technology designed to address this challenge by creating a more direct and efficient data path.

A. NVIDIA GPUDirect Storage (GDS) Architecture and Benefits (Reduced CPU overhead, lower latency)

NVIDIA GPUDirect Storage enables a direct data path for Direct Memory Access (DMA) transfers between GPU memory and storage systems, whether local (e.g., NVMe SSDs) or remote (e.g., network-attached parallel file systems).[57] This architecture fundamentally changes how GPUs access data by avoiding the traditional path where data is first copied from storage into CPU system memory (a "bounce buffer") and then from CPU memory to GPU memory.

The primary benefits of GDS include [54]:

- **Increased System Bandwidth:** By eliminating the CPU memory bottleneck, GDS allows for higher data transfer rates between storage and GPUs.
- **Decreased Latency:** Direct transfers reduce the number of steps and system calls involved, leading to lower data access latency.
- **Reduced CPU Utilization:** Offloading data transfer operations from the CPU frees up CPU cycles for other computational tasks or allows for more power-efficient operation.

GDS is a component of the NVIDIA Magnum IO SDK, a suite of software designed to optimize I/O for accelerated data centers.[15] The GPUDirect family of technologies also includes GPUDirect RDMA (for direct GPU-to-NIC communication over a network) and GPUDirect

Peer-to-Peer (for direct GPU-to-GPU communication within the same system or across NVLink).[88] GDS is particularly beneficial in scenarios where I/O is a significant bottleneck and the CPU is heavily utilized in managing data transfers to and from its memory.[54] Storage solutions like MinIO AIStor are integrating GDS support to reduce CPU consumption on GPU servers, thereby improving overall system efficiency.[88]

By bypassing the CPU for data transfers directly to GPU memory, GDS effectively shifts the potential performance bottleneck from CPU-mediated I/O to the inherent limits of the storage system and the network fabric.[54] This underscores the importance of pairing GDS with high-performance storage (like parallel file systems or fast NVMe arrays) and low-latency, high-bandwidth networks. If the CPU is no longer the chokepoint for data movement to the GPU, the speed at which the storage can source data and the network can transmit it become the new limiting factors.

However, GDS is not a universally applicable solution for all I/O operations. It has specific requirements, such as the need for RDMA-capable networks when accessing remote storage, support from the underlying file system, and often the use of O_DIRECT file access mode to bypass the OS page cache.[54] For certain types of I/O, such as operations on very small files, encrypted files, or compressed files where direct DMA is not feasible, GDS may fall back to a "compatibility mode." In this mode, data is transferred through a more traditional path, and the performance benefits of GDS are diminished or lost.[54] Therefore, applications and data formats must often be GDS-aware or GDS-friendly to achieve the maximum performance improvements.

B. The cuFile API: Enabling Direct Storage-to-GPU Transfers

The primary interface for applications and frameworks to leverage GDS is the **cuFile API**.[86] This API is part of the GDS software stack and provides a set of functions that allow CUDA applications to perform high-performance I/O directly between storage and GPU memory.

Key functionalities of the cuFile API include [87]:

- **File Registration:** cuFileHandleRegister registers a file descriptor with the GDS driver, preparing it for direct I/O operations.
- **Direct Read/Write Operations:** Functions like cuFileRead and cuFileWrite are analogous to POSIX pread and pwrite but are designed to operate with GPU memory buffers as the source or destination. These calls initiate DMA transfers directly between the storage device and the specified GPU memory region.
- **Stream Association:** cuFile operations can be associated with CUDA streams using cuFileReadAsync and cuFileWriteAsync. This enables asynchronous I/O, where data transfers are ordered with respect to computations on a CUDA stream, allowing for effective overlapping of I/O and computation.
- **Batch Operations:** APIs like cuFileReadBatch and cuFileWriteBatch allow for submitting multiple I/O requests in a single call, which can amortize overhead and improve efficiency for many small I/Os.

The underlying mechanism often involves the nvidia-fs.ko kernel module, which orchestrates the direct I/O from DMA/RDMA-capable storage devices to the user-allocated GPU memory.[89] If a direct path is not possible due to unsupported configurations or file types, the

cuFile library can transparently fall back to a compatibility mode, typically involving staging data through CPU system memory, ensuring that the APIs can be used ubiquitously even if GDS acceleration is not available for a particular operation.[87]

The cuFile API represents a significant shift towards GPU-centric I/O programming. It empowers developers to explicitly manage and optimize data transfers between storage and GPU memory, treating the GPU as a first-class participant in I/O operations rather than a passive recipient of data shuttled through the CPU. This explicit, proactive approach to data movement, as opposed to implicit requests triggered by page faults, is key to maximizing performance in GDS-enabled systems.[87] This paradigm requires modifications in how I/O is handled within ML frameworks and custom applications to fully exploit GDS capabilities.

C. GDS Integration with Parallel File Systems (e.g., Lustre with GDS, IBM Spectrum Scale with GDS)

For GDS to be effective with the large, persistent datasets typically used in ML training, the underlying storage systems, particularly parallel file systems, must provide native integration. This involves kernel drivers and user-space libraries that can map file I/O operations to GDS direct transfers.

- **Lustre:** Amazon FSx for Lustre is a prominent example of a managed Lustre service that supports GDS. When used with EFA-capable EC2 instances, FSx for Lustre can leverage GDS to enable direct data transfer between the file system and GPU memory, achieving very high client throughput (up to 1200 Gbps per client is reported).[49] GDS support is often automatically enabled on EFA-enabled FSx for Lustre file systems when

accessed from appropriately configured clients.[50]

- **IBM Spectrum Scale (GPFS):** This parallel file system also provides robust GDS support. It allows data to be read or written directly from an NSD (Network Shared Disk) server's pagepool to the GPU buffers on client nodes via RDMA (over InfiniBand or RoCE).[45] This requires the CUDA toolkit to be installed on the GDS clients and appropriate MOFED (Mellanox OpenFabrics Enterprise Distribution) drivers for the RDMA fabric.
- **Other File Systems:** The nvidia-fs.ko driver, which is a core component of the GDS software stack, also lists support for other file systems such as XFS and EXT4 (when used in ordered mode on NVMe/NVMeOF devices), NFS over RDMA (with MOFED 5.1 and above), and other RDMA-capable distributed file systems like DDN EXAScaler (which underlies some Lustre distributions), WekaFS, and VAST Data.[89] Quantum's Myriad all-flash file system is also developing a client that leverages GDS technology.[59]

This broad support across various high-performance file systems indicates that GDS is becoming a standard feature for storage solutions targeting AI/ML workloads. The file system itself must cooperate with the cuFile library and the GDS kernel components (like nvidia-fs.ko or the newer upstream kernel PCI P2PDMA infrastructure [87]) to enable the direct data path.

D. GDS in ML Frameworks: PyTorch (DALI, KvikIO) and TensorFlow (DALI)

While GDS provides the low-level infrastructure for direct GPU-storage I/O, its benefits are most readily realized when integrated into high-level ML frameworks. This integration is still evolving.

- **PyTorch:**

    - The default PyTorch Dataset and DataLoader classes primarily work with standard POSIX file APIs for data loading and checkpointing.[91]
    - To leverage GDS with PyTorch, specialized libraries are often required:
        - **NVIDIA DALI (Data Loading Library):** DALI is a library designed to accelerate data loading and preprocessing pipelines by offloading these tasks to the GPU. DALI can replace or augment built-in PyTorch DataLoaders and provides support for GDS for certain data formats (e.g., NumPy arrays) when reading from file-based storage.[91]
        - **KvikIO:** An open-source library from RAPIDS that provides Python and C++ bindings directly to the cuFile API, enabling GDS access. KvikIO can be integrated into PyTorch data loading pipelines to read/write data directly to/from GPU memory.[91]
    - There is active interest and feature requests within the PyTorch community for more first-class, native support for GDS within core components like IterableDataset and for checkpointing operations, aiming to simplify its adoption and broaden its applicability.[95]
- **TensorFlow:**
    - TensorFlow's standard data input pipeline API is tf.data.
    - Similar to PyTorch, direct GDS integration into the core tf.data mechanisms is not as explicit. However, NVIDIA DALI also offers a TensorFlow plugin (nvidia-dali-tf-plugin) that allows DALI pipelines (which can include GDS-accelerated operations and GPU-based augmentations) to be seamlessly integrated as a data source for TensorFlow models.[92]
    - While older discussions touch upon GPU memory

management and data prefetching in TensorFlow [97], and the DALI TensorFlow plugin API documentation exists [98], the specifics of tf.data directly using cuFile without an intermediary like DALI are less clear from the provided materials.

The current state suggests that GDS adoption within major ML frameworks, while progressing, often relies on intermediary libraries like DALI or direct integration of cuFile wrappers like KvikIO. This adds a layer of complexity for ML engineers wishing to leverage GDS, as it may require deviating from standard framework data loaders or incorporating additional dependencies. NVIDIA DALI, in particular, is emerging as a key enabler, acting not only as a GPU-accelerated data augmentation library but also as a bridge to GDS capabilities for both PyTorch and TensorFlow.[91] This makes DALI a critical component for building high-performance input pipelines that can fully exploit GDS.

E. Best Practices for Staging Preprocessed Data for GDS Access

To maximize the benefits of GPUDirect Storage, data should be appropriately prepared and "staged" on storage systems that are GDS-accessible and in a format conducive to direct transfers.

1. **Separate Preprocessing and Staging:** Preprocessing tasks (cleaning, transformation, augmentation) are often CPU-intensive. It's a good practice to perform these using distributed CPU-driven systems like Apache Spark or Ray Data. The results of this preprocessing should then be written (staged) to a GDS-enabled, high-performance storage system (e.g., a parallel file system like Lustre or Spectrum Scale).[88]
2. **Use High-Performance GDS-Compatible Storage:** Pair GDS

with storage solutions that can match its throughput potential, such as NVMe SSDs, NVMe-over-Fabrics (NVMe-oF) arrays, or high-performance parallel file systems specifically validated for GDS compatibility.[56]

3. **Optimize Data Format and Layout:** GDS, particularly when using O_DIRECT for bypassing the OS cache, performs best with large, contiguous, and memory-aligned I/O operations.[87] This might influence how datasets are sharded, batched, or serialized during the preprocessing stage. Avoid many small, random I/O operations if possible. Uncompressed data often allows for the most direct GDS path.
4. **Leverage Tiered Storage:** Implement a tiered storage strategy where the actively used, preprocessed training data resides on the GDS-enabled hot tier, while raw or less frequently accessed data is kept on slower, more cost-effective tiers.[56]
5. **Profile and Monitor I/O:** Use profiling tools (like NVIDIA's gdsio utility [57] or other system monitoring tools) to understand I/O patterns, identify bottlenecks, and verify that GDS is being effectively utilized.[56] Check for GDS compatibility mode fallbacks.[54]
6. **Integrate with DataLoader APIs:** When using ML frameworks, ensure that the DataLoader APIs are configured to efficiently read batches of the staged, preprocessed data from the GDS-enabled storage in real-time to feed the GPUs.[88]

The "staging" step is critical because GDS primarily accelerates the *transfer* of data that is already in a suitable state and location. It's not a magic bullet that makes any data source instantly fast for GPU access. The data must first be transformed and placed onto a storage system that GDS can efficiently interact with. This implies a

deliberate multi-stage data pipeline where raw data is ingested, processed, and then explicitly staged for GDS-accelerated consumption by the training cluster. This careful preparation ensures that the high-speed path offered by GDS can be fully exploited. The preference of GDS for larger, contiguous file access patterns also suggests that data engineering practices upstream of model training should consider organizing data into larger chunks or files, favoring sequential access patterns to maximize GDS efficiency.[87]

VII. MLOps: Ensuring Reproducibility, Efficiency, and Governance

Machine Learning Operations (MLOps) encompasses the practices, tools, and cultural shifts required to build, deploy, and maintain ML systems reliably and efficiently at scale. For a modern ML system, a robust MLOps framework is indispensable for managing the complexities of the ML lifecycle, including experiment tracking, artifact management, model versioning, hyperparameter optimization, and model serving. Key goals are to ensure reproducibility, enhance collaboration, improve efficiency, and provide governance.[25]

A. Experiment Tracking and Artifact Management (e.g., MLflow, Kubeflow, DVC – backend/artifact store choices)

Tracking ML experiments—including parameters, metrics, code versions, and generated artifacts—is fundamental for reproducibility, debugging, and comparing different modeling approaches.[24]

- **MLflow:** An open-source platform designed to manage the end-to-end ML lifecycle.[25]
    - **MLflow Tracking:** Allows logging of parameters, metrics, source code versions (if using MLflow Projects), and output artifacts for each experimental run. It provides a UI for

visualizing and comparing runs.[26]

- **Backend Store:** Persists the lightweight metadata associated with runs (e.g., run ID, parameters, metrics, tags). MLflow supports file-system-based backends (storing metadata in local files, typically within a ./mlruns directory) or, for more scalable and collaborative setups, database-backed stores such as PostgreSQL, MySQL, or SQLite.[26] Using a database backend is a requirement for leveraging MLflow Model Registry features.[71]
- **Artifact Store:** Stores the larger output files (artifacts) from runs, such as trained model files, data samples, or visualizations. MLflow supports various artifact stores including local file paths (e.g., on an NFS mount), Amazon S3, Azure Blob Storage, Google Cloud Storage, and HDFS.[8] S3-compatible object stores like MinIO or OVHcloud Object Storage can also be used.[104]
- *Implications for GDS with MLflow:* If MLflow's artifact store is configured to use a GDS-enabled file system (e.g., Lustre or Spectrum Scale, potentially mounted via NFS or accessed through a custom URI handler if MLflow supports it), then artifacts written to or read from this store by GDS-aware applications could benefit. For instance, if a training job saves a large model checkpoint (an artifact) to a GDS-enabled Lustre file system, and a subsequent evaluation job (also GDS-aware) reads this checkpoint, the transfer could be accelerated. If the artifact store is an object store like S3, direct GDS benefits are less likely unless an S3 gateway with GDS support or a GDS-enabled caching layer (like Alluxio) is placed in front of the S3 bucket. MLflow primarily records the artifact_uri, pointing to the artifact's

location.

- **Kubeflow Pipelines (KFP):** A component of Kubeflow for building, deploying, and managing multi-step ML workflows (pipelines) on Kubernetes.[24]
  - **Metadata Store:** KFP stores metadata about pipeline runs, experiments, jobs, and individual pipeline step inputs/outputs. This is typically stored in a MySQL database deployed as part of Kubeflow.[105]
  - **Artifact Store:** KFP stores pipeline artifacts (which can include serialized data, models, metrics files, visualizations) in an object store. Supported backends include MinIO (often deployed by default with Kubeflow), Amazon S3, and Google Cloud Storage.[105] The location for these artifacts is configured via the pipeline_root setting, which can be specified at the pipeline definition level or when submitting a run.[107]
  - *Implications for GDS with KFP:* Similar to MLflow, if the KFP pipeline_root points to an object store bucket that is, for example, an S3 gateway to a GDS-enabled parallel file system, or if KFP components within pipeline steps are written to be GDS-aware when interacting with a directly mounted GDS-enabled file system (if KFP allows such file:// URIs for artifacts), then GDS could accelerate artifact I/O. The directness of this integration depends on KFP's artifact handling mechanisms and whether pipeline components can leverage cuFile.
- **DVC (Data Version Control):** An open-source tool that versions data and models by storing small metadata files (containing checksums and pointers) in Git, while the actual large data files are stored in a separate remote storage location.[9] Supported

remotes include S3, GCS, Azure Blob Storage, SSH servers, HDFS, and local file systems.

- *Implications for GDS with DVC:* If DVC's configured remote storage is a GDS-enabled parallel file system, then dvc push would store data there, and dvc pull would retrieve it. If the local workspace where dvc pull materializes the data (or a local cache directory used by DVC) is on a GDS-enabled file system (e.g., local NVMe with GDS support, or a GDS-enabled scratch space), then subsequent training jobs that are GDS-aware could access this data with GDS acceleration.

A common architectural pattern observed in tools like MLflow and Kubeflow is the decoupling of lightweight metadata storage from heavyweight artifact storage.[8] Metadata, which includes parameters, metrics, and run information, benefits from the querying and transactional capabilities of a database. Artifacts, such as large model files or datasets, require scalable and often more cost-effective bulk storage solutions like object stores or parallel file systems. This separation allows for independent optimization of each storage type.

The choice of artifact store has significant implications for the performance of downstream tasks. If preprocessed datasets or trained models, managed as artifacts by these MLOps tools, are stored on high-performance, GDS-enabled storage, subsequent pipeline steps like further training, model evaluation, or model serving can access these artifacts much more rapidly. In such scenarios, the artifact store transitions from being a passive repository to an active component of the high-performance data pipeline, with the artifact_uri (in MLflow) or pipeline_root (in

Kubeflow) becoming a critical path for overall system performance.

B. Model Versioning Strategies for Large Models and Datasets (e.g., MLflow Model Registry, DVC with remote storage)

Effective versioning in ML requires tracking not only the model binary itself but also the training code, the specific dataset version used, hyperparameters, and the software environment to ensure full reproducibility.[99]

- **MLflow Model Registry:** Provides a centralized repository for managing the lifecycle of MLflow Models.[112] It allows users to:
  - Register models that have been logged during MLflow Tracking runs.
  - Version these registered models (e.g., "Version 1", "Version 2").
  - Assign stages to model versions (e.g., "Staging", "Production", "Archived").
  - Use aliases (e.g., "champion") to point to specific model versions for easier reference in deployment.
  - Add tags and annotations for better organization and description.
  - Track model lineage, linking a model version back to the MLflow run that produced it (which contains information about parameters, metrics, and source code).[114] Access to the Model Registry typically requires a database-backed backend store for MLflow.[71] Models can be retrieved for deployment using URIs like models:/<model_name>/<model_version> or models:/<model_name>/<alias>.[114]
- **DVC (Data Version Control):** As previously discussed, DVC versions large files, including models and datasets, by storing

their checksums and metadata in Git, while the actual files reside in remote storage.[9] The workflow involves using dvc add <model_file_or_data_dir> to track changes, dvc commit to record the new version's metadata, and git commit to version the .dvc files. dvc push and dvc pull are used to synchronize the actual large files with the configured remote storage. This approach tightly couples data and model versions with code versions in Git.

- **Git LFS (Large File Storage):** An extension to Git designed to handle large binary files more efficiently than native Git.[9] Git LFS replaces large files in the Git repository with small text pointer files. The actual large files are stored on a separate Git LFS server. While simpler than DVC for basic large file versioning alongside code, Git LFS is less specialized for the broader ML context of tracking experiments, metrics, or complex data dependencies.

True model reproducibility extends beyond merely versioning the model file itself. It necessitates capturing the entire context of the model's creation: the exact version of the training code, the specific dataset snapshot used, all hyperparameters, and the complete software environment (libraries, drivers, OS).[99] Tools like MLflow achieve this by linking registered models back to the comprehensive tracking data of the run that produced them. DVC achieves this by enabling the versioning of data, code, and pipeline definitions together within a Git repository.

Furthermore, modern ML "models," especially complex ones like LLMs, are often not single files but collections of artifacts. These can include model weights, tokenizer files, configuration files, and even prompt templates. Versioning systems must be capable of handling

these grouped artifacts cohesively as a single versionable "model unit." MLflow's concept of an "MLflow Model" (a directory containing the model files and a MLmodel descriptor file) inherently supports this.[114] DVC can track entire directories, allowing a collection of related model files to be versioned together.

C. Distributed Hyperparameter Optimization (e.g., Ray Tune, Optuna – data access patterns)

Hyperparameter Optimization (HPO) is a critical step in maximizing model performance. Automating and scaling HPO can significantly accelerate the model development process.

- **Ray Tune:** A distributed HPO library that is part of the Ray ecosystem.[103] Ray Tune can launch multiple HPO trials concurrently across a Ray cluster, leveraging multiple nodes and GPUs.[119] It integrates seamlessly with Ray Train, allowing each HPO trial to itself be a distributed training run.[120] Ray Tune supports a variety of advanced search algorithms (e.g., ASHA, HyperBand, Population Based Training) and early stopping techniques to efficiently explore the hyperparameter space.[103] Trial states and checkpoints can be saved to persistent storage, including cloud object storage like S3, which is important for fault tolerance and resuming long-running HPO jobs.[121]
- **Optuna:** A lightweight yet powerful HPO framework known for its define-by-run API and intelligent sampling strategies, such as the Tree-structured Parzen Estimator (TPE).[103] Optuna can be parallelized for distributed HPO in several ways: using Joblib with a Spark backend for distribution across a Spark cluster [122], or by using a shared relational database (e.g., PostgreSQL) to store trial states and coordinating multiple Optuna worker processes, often orchestrated by Kubernetes.[123] Optuna also

integrates with MLflow for logging and tracking HPO trials.[122]

During distributed HPO, each trial typically requires access to the training and validation datasets. If these datasets are large and stored on shared infrastructure (e.g., NFS, a parallel file system, or object storage with a caching layer), the storage system must be able to handle concurrent access from many trials without becoming a bottleneck. When Ray Tune is used with Ray Data, data can be streamed and preprocessed efficiently for each trial.[82] If data resides in cloud object storage like S3, as is common with Ray Tune setups [121], each trial or its group of distributed workers will fetch the data. In such scenarios, data caching layers (e.g., Alluxio, JuiceFS) or efficient data loading mechanisms within each trial become crucial to prevent HPO from being I/O-bound.

The tight integration of HPO tools with distributed training frameworks, exemplified by Ray Tune and Ray Train [120], marks an important trend. HPO is evolving from simply launching many independent, single-node training jobs to orchestrating and managing multiple *distributed* training jobs as individual HPO trials. This requires the HPO system to have more sophisticated resource management capabilities, allocating resources (CPUs, GPUs) for each distributed trial, overseeing its lifecycle, and aggregating results. This close coupling enables more advanced HPO strategies to be applied effectively to complex, distributed ML models.

D. Considerations for Model Serving (e.g., KServe, Triton with MLflow integration)

Deploying trained and versioned models for inference is the final step in delivering value from the ML system. Serving platforms need to be scalable, reliable, and capable of loading specific model versions.

- **MLflow:** Offers built-in capabilities for model deployment. The mlflow models serve command can deploy an MLflow Model as a local REST API endpoint, using FastAPI (default) or MLServer as the backend serving engine.[115] The MLServer backend is particularly significant as it enables integration with Kubernetes-native serving frameworks like KServe (formerly KFServing) and Seldon Core, allowing MLflow Models to be deployed in scalable, production-grade Kubernetes environments.[115] Models are typically retrieved from the MLflow Model Registry using URIs that specify the model name and version (e.g., models:/<model_name>/<model_version>) or an alias.[114]
- **KServe:** A standard Model Inference Platform on Kubernetes, built for highly scalable and production-ready model serving. It is often used as the serving component within Kubeflow.[24] KServe supports features like serverless inference, scale-to-zero, canary deployments, and explainability.
- **NVIDIA Triton Inference Server:** A high-performance inference server that supports models from various ML frameworks (TensorFlow, PyTorch, ONNX, TensorRT, etc.). Triton is designed for maximizing throughput and utilization on GPUs (and CPUs). It can be deployed standalone or integrated with platforms like KServe.

For a seamless MLOps workflow, the model serving system must integrate with the model versioning and artifact management components. Typically, a CI/CD pipeline would trigger the deployment of a new model version to the serving environment once it has been validated and promoted (e.g., to "Production" stage in MLflow Model Registry). The serving platform then fetches the

specified model artifacts from the artifact store (e.g., S3, NFS) based on the information provided by the model registry.

The MLflow Model Registry plays a pivotal role in this CI/CD process for models.[114] By versioning models and tracking their lifecycle stages, it provides the necessary control and traceability for automated deployment, A/B testing, and rollbacks. Serving tools are configured to pull specific, approved model versions from this registry, ensuring that the correct model is deployed into production.

Furthermore, the standardization of model packaging is key for achieving interoperable and flexible model serving. MLflow's "MLflow Model" format, which packages a model along with its dependencies and a standardized descriptor file (MLmodel), aims to provide a common format that can be understood and deployed by various serving tools (MLflow's own server, MLServer, Amazon SageMaker, etc.).[114] This reduces the friction involved in deploying models trained in different ML frameworks, as the serving infrastructure can rely on the standardized MLflow format rather than needing to intimately understand the specifics of every framework.

VIII. System Integration: Connecting the Components for a Cohesive ML Platform

A modern large-scale ML system is not a monolithic entity but a complex ecosystem of specialized sub-systems for compute, networking, storage, data processing, and MLOps. Effective integration of these components is crucial for building a cohesive and high-performing platform.

A. Overview of Component Interactions (Visualized for Mermaid)

The following outlines the primary data and control flows within the

proposed ML system architecture. This description is intended to be suitable for generating a Mermaid diagram to visually represent these interactions.

Data Flow:

1. **Raw Data Ingestion:**
    - Source: External data sources (databases, APIs, logs, existing data lakes).
    - Destination: **Warm Tier Object Storage** (e.g., S3-compatible like MinIO). This serves as the primary, scalable data lake for raw and semi-processed data.
    - Mechanism: Ingestion pipelines, potentially using Kafka for streaming or batch tools for bulk loads.
2. **Data Preprocessing:**
    - Source: Warm Tier Object Storage.
    - Processing: **Distributed Preprocessing Cluster** (e.g., Apache Spark, Ray Data, or Dask running on CPU-optimized Kubernetes nodes or dedicated CPU cluster).
    - Caching (Optional): If object storage is the source, a caching layer like **Alluxio** or **JuiceFS** can sit between object storage and the preprocessing cluster to accelerate reads.
    - Destination (Staged Data): **Hot Tier Parallel File System** (e.g., Lustre, IBM Spectrum Scale, BeeGFS), which should be GDS-enabled. Alternatively, for smaller datasets or specific Ray Data workflows, data might be directly streamed or cached within the Ray cluster's memory/local disk.
    - Mechanism: Preprocessing frameworks read from object storage (via cache if present), perform transformations, and write results to the parallel file system.
3. **Model Training Data Path:**

    - Source: Hot Tier Parallel File System (or Alluxio/JuiceFS cache over object storage if GDS is integrated there).
    - Destination: **GPU Memory** within the GPU Training Cluster.
    - Mechanism: High-Performance Network Fabric (e.g., InfiniBand or Ethernet with RoCEv2). Data loaders within ML frameworks (e.g., PyTorch DataLoader, TensorFlow tf.data, potentially using NVIDIA DALI or KvikIO) read data from the parallel file system, leveraging **NVIDIA GPUDirect Storage (GDS)** and the **cuFile API** for direct DMA transfer into GPU memory, bypassing the CPU.
4. **Model Checkpointing:**
    - Source: GPU Memory (model state during training).
    - Destination: **Hot Tier Parallel File System** or a **High-Performance NFS Server** (optimized for writes, possibly with RDMA).
    - Mechanism: Training script periodically saves model checkpoints. GDS can also accelerate writes if the target storage is GDS-enabled and the framework supports GDS for checkpointing.
5. **Model Artifacts & Experiment Logging:**
    - Source: Training scripts, evaluation scripts.
    - Destination (Artifacts): **MLOps Artifact Store** (e.g., Object Storage, NFS, or even the parallel file system, as configured in MLflow/Kubeflow).
    - Destination (Metadata): **MLOps Backend Store** (e.g., PostgreSQL database for MLflow/Kubeflow).
    - Mechanism: MLOps tools (MLflow, Kubeflow Pipelines) log metrics, parameters to the backend store and save model files, visualizations, etc., to the artifact store.

Control/Orchestration Flow:

1. **User/CI-CD System Interaction:**
    - Interface: Git (for code, DVC metafiles), MLOps Platform UI/CLI (MLflow, Kubeflow).
    - Action: User commits code, triggers pipeline, launches experiment, promotes model.
2. **MLOps Platform Orchestration:**
    - Component: **MLOps Orchestrator** (e.g., MLflow Projects, Kubeflow Pipelines, custom scripts invoking Ray/Slurm jobs via Kubernetes).
    - Action 1 (Preprocessing): Schedules and manages data preprocessing jobs on the Distributed Preprocessing Cluster.
    - Action 2 (Training): Schedules and manages model training jobs on the **GPU Training Cluster** (which itself is managed by a lower-level orchestrator like Kubernetes, Slurm, or Ray). Passes hyperparameters, data paths.
    - Action 3 (Tracking/Versioning): Interacts with MLOps Backend Store and Artifact Store to record experiment details and version models/data (e.g., via MLflow Tracking Server, MLflow Model Registry, DVC commands).
3. **GPU Cluster Workload Management:**
    - Component: **Cluster Manager** (Kubernetes, Slurm, Ray).
    - Action: Allocates GPU and other resources to training jobs, manages container execution (if applicable), monitors job status.
4. **Model Serving Deployment:**
    - Source: **MLOps Model Registry** (e.g., MLflow Model Registry).
    - Action: CI/CD pipeline or manual trigger initiates deployment

of a specific model version.
    - Destination: **Model Serving Platform** (e.g., KServe, NVIDIA Triton Inference Server, running on Kubernetes or dedicated inference cluster).
    - Mechanism: Serving platform retrieves the specified model artifacts from the MLOps Artifact Store (location obtained from the Model Registry) and deploys the model as an inference endpoint.

This interconnectedness highlights that the system is far more than the sum of its parts. No single technology or component dominates all aspects of the ML lifecycle. For instance, object storage is excellent for establishing a scalable raw data lake, but high-performance parallel file systems are indispensable for the demanding I/O patterns of model training. Similarly, CPU-based clusters are optimal for data preprocessing tasks, while GPU clusters are the workhorses for training deep learning models. This inherent specialization across different stages necessitates careful and robust integration between these sub-systems.

Furthermore, network performance is not just a concern for GPU-to-GPU communication within the training cluster.[5] It is equally critical at multiple other interfaces.[88] The network link between the primary data lake (object storage) and the preprocessing cluster, the link from the preprocessing cluster to the staging storage (parallel file system), and finally, the fabric connecting the staging storage to the GPU training cluster—all these data paths must be adequately provisioned in terms of bandwidth and latency. A bottleneck in any of these segments can starve the GPUs and negate the benefits of a high-performance training cluster.

B. Example Technology Stack and Configuration Notes (Illustrative choices for each layer)

To provide a more concrete illustration, the following table outlines an example technology stack for a modern ML system, combining many of the components discussed. This is an illustrative example, and specific choices would depend on budget, scale, existing infrastructure, and team expertise.

Table 4: Example ML System Technology Stack

System Layer	Specific Technology Choice	Key Configuration Notes/Rationale
GPU Compute Nodes	Servers with 8x NVIDIA H200 141GB GPUs	SXM form factor for high intra-node bandwidth via NVLink/NVSwitch. Paired with high-core count CPUs (e.g., AMD Epyc or Intel Xeon Scalable) and ample system RAM (e.g., 1-2TB per node).
GPU Cluster Interconnect	NVIDIA Quantum-2 InfiniBand (NDR 400Gbps) or 800GbE Ethernet with RoCEv2 (e.g., Arista 7060X6/7800R4 series [5])	2-level Fat-Tree (Leaf-Spine) topology. Ensure RDMA is enabled. For RoCEv2: configure PFC, ECN for lossless operation; ensure consistent MTU

		across fabric.
Cluster Orchestration	Kubernetes (e.g., GKE, EKS, AKS, or on-prem) with Ray deployed on Kubernetes (Ray Operator)	Kubernetes for base infrastructure management. Ray for distributed Python/ML workloads (Ray Train, Ray Tune, Ray Serve). GPU sharing (MIG) and node labeling in Kubernetes.[7]
Storage - Hot Tier (Training Data)	Lustre file system (e.g., managed cloud service like AWS FSx for Lustre [49] or on-prem DDN EXAScaler)	GDS-enabled for direct GPU access. High-throughput configuration (e.g., SSD-based). POSIX access for frameworks.
Storage - Warm Tier (Raw Data Lake, Large Artifacts)	S3-compatible Object Storage (e.g., MinIO, Ceph RGW, or cloud provider S3/GCS/Azure Blob)	Highly scalable and durable. Cost-effective for large volumes.
Storage - Caching/Orchestration for Warm Tier	Alluxio or JuiceFS	Deployed between object storage and compute (preprocessing/training clusters). Provides POSIX access and distributed caching to

		accelerate reads from object storage.[63]
Storage - Scratch (Node-Local)	Local NVMe SSDs on GPU and CPU nodes	Managed via ZFS LocalPV in Kubernetes for dynamic provisioning of scratch volumes.[43] Used for temporary data, shuffle operations.
Storage - MLOps Artifacts/Checkpoints (Alternative)	High-Performance NFS Server (e.g., NetApp, Dell PowerScale, or custom build with NFS over RDMA [46])	For MLflow artifact store if not using object storage directly. Optimized for mixed I/O, reliable for checkpoints.
Data Ingestion	Apache Kafka + Apache Spark (running on Kubernetes)	Kafka for real-time stream ingestion and buffering.[77] Spark for batch ETL from diverse sources and initial processing.[74]
Data Preprocessing (Final Stage)	Ray Data (running on Ray cluster within Kubernetes)	For scalable, distributed preprocessing feeding directly into Ray Train. Leverages CPU nodes efficiently.[81]

MLOps - Experiment Tracking & Model Registry	MLflow Tracking Server	Backend: Managed PostgreSQL database.[71] Artifact Store: S3-compatible Object Storage (via Alluxio/JuiceFS if caching needed) or the dedicated NFS server.[8]
MLOps - Data & Model Versioning	DVC (Data Version Control)	Integrated with Git. Remote storage for DVC cache pointing to the S3-compatible object store or parallel file system.
MLOps - Hyperparameter Optimization	Ray Tune	Integrated with Ray Train. Leverages the Ray cluster for distributed trials. Checkpoints trials to shared storage (e.g., NFS or object store).[121]
MLOps - Model Serving	KServe on Kubernetes, with NVIDIA Triton Inference Server as the backend	Models pulled from MLflow Model Registry. KServe for scalable, standardized deployment.[115]

A hybrid orchestration model, using Kubernetes as the foundational platform with Ray deployed on top for ML-specific workloads, offers

considerable flexibility. Kubernetes handles the underlying infrastructure provisioning, scaling, and management of containerized applications, including Ray clusters themselves (often via a Ray Kubernetes Operator). Ray then provides the specialized environment and libraries (Ray Train, Ray Tune, Ray Data, Ray Serve) tailored for distributed Python and machine learning tasks.[6] For certain highly parallel, HPC-style training jobs, an orchestrator like Slurm might even be run on Kubernetes (e.g., via Soperator [6]) to leverage its advanced batch scheduling capabilities. This layered approach combines the strengths of general-purpose container orchestration with domain-specific ML and HPC workload management.

IX. Conclusion and Future Outlook

The design of a modern, large-scale ML system is a multifaceted endeavor, requiring a delicate balance between raw component performance, system-level integration, and operational efficiency. The blueprint outlined in this report emphasizes several core principles: the synergistic performance of compute, network, and storage; the necessity of high-speed, low-latency networking with RDMA capabilities; a tiered and specialized storage architecture to cater to diverse data access patterns; optimized data paths directly to GPU memory via technologies like GPUDirect Storage; and a comprehensive MLOps framework to ensure reproducibility, governance, and agility throughout the ML lifecycle. Building such a system is undoubtedly complex, but it is a prerequisite for organizations aiming to stay at the cutting edge of AI research and deployment.

Looking ahead, several trends are poised to further shape the

evolution of ML systems:

1. **Deeper Integration of Compute, Network, and Storage:** The lines between these traditionally distinct domains will continue to blur. Technologies like Data Processing Units (DPUs) that offload networking and storage tasks from CPUs, computational storage devices that perform processing directly on stored data, and increasingly intelligent network fabrics (as envisioned by the UEC and exemplified by NVIDIA SHARP [16]) point towards a future where system components are more deeply aware of and integrated with each other. This tighter coupling aims to minimize data movement and process data closer to where it resides or where it is needed, further reducing latency and improving efficiency.
2. **Advancements in Hardware/Software Co-design:** The development of AI-specific hardware (next-generation GPUs, custom ASICs) will increasingly be accompanied by co-designed software stacks (libraries, compilers, frameworks) to extract maximum performance.[57] This co-design philosophy ensures that software can fully exploit unique hardware features, and hardware is architected with the needs of leading AI workloads in mind.
3. **More Intelligent and Automated MLOps:** MLOps platforms will become more sophisticated, incorporating AI itself to automate tasks like optimal resource allocation, proactive anomaly detection in model performance, automated retraining triggers, and intelligent data tiering. The goal is to create self-optimizing and self-healing ML pipelines that require less manual intervention.
4. **Sustainability and Total Cost of Ownership (TCO) as**

Primary Design Drivers: As ML clusters scale to unprecedented sizes, their energy consumption and overall TCO are becoming critical concerns.[10] Future designs will place a greater emphasis on power-efficient hardware components, energy-aware scheduling, optimized cooling solutions, and software techniques that maximize resource utilization to reduce idle power. The development of more heterogeneous compute environments, leveraging different types of accelerators for different tasks, may also contribute to better energy efficiency.

The journey to build and operate these advanced ML systems requires continuous learning, adaptation, and a commitment to a holistic, integrated design approach. The principles and technologies discussed provide a robust foundation for architecting the ML infrastructure of tomorrow.

Works cited

1. Machine Learning Architecture: What It Is, Components & Types, accessed May 19, 2025, https://lakefs.io/blog/machine-learning-architecture/
2. GPU Cluster Explained: Architecture, Nodes and Use Cases - Scale Computing, accessed May 19, 2025, https://www.scalecomputing.com/resources/what-is-a-gpu-cluster
3. Scalable Architecture Patterns for High-Growth Startups That Every Business Owner Should Know Today - Full Scale, accessed May 19, 2025, https://fullscale.io/blog/scalable-architecture-patterns/
4. How to Build Great Machine Learning Infrastructure - Anyscale, accessed May 19, 2025, https://www.anyscale.com/glossary/ml-machine-learning-infrastructure
5. AI Networking Center | Artificial Intelligence AI technology - Arista, accessed May 19, 2025, https://www.arista.com/en/solutions/ai-networking
6. What are GPU clusters and how to choose yours? - Nebius, accessed May 19, 2025, https://nebius.com/blog/posts/what-are-compute-clusters-and-how-to-choose-yours
7. Kubernetes GPU Resource Management Best Practices - PerfectScale, accessed May 19, 2025, https://www.perfectscale.io/blog/kubernetes-gpu
8. Artifact Stores | MLflow, accessed May 19, 2025,

https://mlflow.org/docs/latest/tracking/artifacts-stores/
9. What is the best way to do data version control for Machine Learning models - MATLAB Answers - MathWorks, accessed May 19, 2025, https://www.mathworks.com/matlabcentral/answers/2067961-what-is-the-best-way-to-do-data-version-control-for-machine-learning-models
10. What are the best practices for configuring NVIDIA H100 GPUs for machine learning workloads? - Massed Compute, accessed May 19, 2025, https://massedcompute.com/faq-answers/?question=What%20are%20the%20best%20practices%20for%20configuring%20NVIDIA%20H100%20GPUs%20for%20machine%20learning%20workloads?
11. GPU machine types | Compute Engine Documentation - Google Cloud, accessed May 19, 2025, https://cloud.google.com/compute/docs/gpus
12. Best practices for competitive inference optimization on AMD Instinct™ MI300X GPUs, accessed May 19, 2025, https://rocm.blogs.amd.com/artificial-intelligence/LLM_Inference/README.html
13. AMD Instinct MI300X workload optimization — ROCm Documentation, accessed May 19, 2025, https://rocm.docs.amd.com/en/latest/how-to/rocm-for-ai/inference-optimization/workload.html
14. How do I properly install and configure the NVIDIA A100 GPU in a data center environment?, accessed May 19, 2025, https://massedcompute.com/faq-answers/?question=How%20do%20I%20properly%20install%20and%20configure%20the%20NVIDIA%20A100%20GPU%20in%20a%20data%20center%20environment?
15. Magnum IO Software Developer Kit (SDK), accessed May 19, 2025, https://developer.nvidia.com/magnum-io
16. Magnum IO Software Stack for Accelerated Data Centers - NVIDIA, accessed May 19, 2025, https://www.nvidia.com/en-us/data-center/magnum-io/
17. NVLink & NVSwitch: Fastest HPC Data Center Platform | NVIDIA, accessed May 19, 2025, https://www.nvidia.com/en-us/data-center/nvlink/
18. An Overview of NVIDIA NVLink - FS.com, accessed May 19, 2025, https://www.fs.com/de-en/blog/an-overview-of-nvidia-nvlink-2918.html
19. About GPUs in Google Kubernetes Engine (GKE), accessed May 19, 2025, https://cloud.google.com/kubernetes-engine/docs/concepts/gpus
20. Slurm management system - Introduction - Together AI, accessed May 19, 2025, https://docs.together.ai/docs/slurm
21. Understanding Slurm GPU Management - Run:ai, accessed May 19, 2025, https://www.run.ai/guides/slurm/understanding-slurm-gpu-management
22. Distributed Machine Learning on Akash Network With Ray, accessed May 19, 2025, https://akash.network/docs/guides/machine-learning/ray/
23. What is Ray on Databricks?, accessed May 19, 2025, https://docs.databricks.com/gcp/en/machine-learning/ray/
24. Kubeflow vs MLflow vs ZenML: Which MLOps Platform Is the Best ..., accessed

May 19, 2025, https://www.zenml.io/blog/kubeflow-vs-mlflow
25. Top 10 MLOps Tools in 2025 to Streamline Your ML Workflow - Futurense, accessed May 19, 2025, https://futurense.com/uni-blog/top-10-mlops-tools-in-2025
26. MLflow Tracking, accessed May 19, 2025, https://mlflow.org/docs/latest/tracking/
27. RDMA RoCEv2 for AI workloads on Google Cloud, accessed May 19, 2025, https://cloud.google.com/blog/products/networking/rdma-rocev2-for-ai-workloads-on-google-cloud
28. How InfiniBand Enhances Machine Learning and AI Workloads | Orhan Ergun, accessed May 19, 2025, https://orhanergun.net/how-infiniband-enhances-machine-learning-and-ai-workloads
29. RDMA over Converged Ethernet - Wikipedia, accessed May 19, 2025, https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet
30. Exploring Ultra Ethernet for AI Networks - UEC Presentation Video on YouTube - FiberMall, accessed May 19, 2025, https://www.fibermall.com/blog/ultra-ethernet.htm
31. Why InfiniBand vs Ethernet matters for Data Centers? - Voltage Park, accessed May 19, 2025, https://www.voltagepark.com/blog/why-infiniband-vs-ethernet-matters-for-data-centers
32. InfiniBand vs. Ethernet debate intensifies amid AI explosion - The Register, accessed May 19, 2025, https://www.theregister.com/2024/01/24/ai_networks_infiniband_vs_ethernet/
33. How Ultra Ethernet and UALink Enable High-Performance, Scalable AI Networks - Synopsys, accessed May 19, 2025, https://www.synopsys.com/articles/ultra-ethernet-ualink-ai-networks.html
34. Optimizing AI GPU Clusters: Network, Scale, and Connectivity Solutions | NADDOD, accessed May 19, 2025, https://www.naddod.com/blog/optimizing-ai-gpu-clusters-network-and-scale
35. What is the recommended network topology for an NVIDIA GPU-accelerated HPC cluster?, accessed May 19, 2025, https://massedcompute.com/faq-answers/?question=What%20is%20the%20recommended%20network%20topology%20for%20an%20NVIDIA%20GPU-accelerated%20HPC%20cluster?
36. Advanced Networks for Artificial Intelligence and Machine Learning Computing | AFL Hyperscale, accessed May 19, 2025, https://www.aflhyperscale.com/wp-content/uploads/2024/10/Advanced-Networks-for-Artificial-Intelligence-and-Machine-Learning-Computing-White-Paper.pdf
37. Clos network - Wikipedia, accessed May 19, 2025, https://en.wikipedia.org/wiki/Clos_network
38. Dragonfly topology - Glenn K. Lockwood, accessed May 19, 2025, https://www.glennklockwood.com/garden/dragonfly

39. Benefits of Fat-Tree Topology vs Dragonfly Topology in InfiniBand Network for AI and HPC Workloads - Massed Compute, accessed May 19, 2025, https://massedcompute.com/faq-answers/?question=What%20are%20the%20benefits%20of%20using%20a%20fat-tree%20topology%20versus%20a%20dragonfly%20topology%20in%20an%20InfiniBand%20network%20for%20AI%20and%20HPC%20workloads?
40. Automated storage tiering using machine learning | SNIA | Experts on Data, accessed May 19, 2025, https://snia.org/educational-library/automated-storage-tiering-using-machine-learning-2018
41. Intelligent storage system with machine learning - CS229 - Stanford University, accessed May 19, 2025, https://cs229.stanford.edu/proj2016/report/Yen-IntelligentStorageSystemWithMachineLearning-report.pdf
42. Asking for help resolving bottlenecks for a small/medium GPU cluster : r/HPC - Reddit, accessed May 19, 2025, https://www.reddit.com/r/HPC/comments/1kbvxxv/asking_for_help_resolving_bottlenecks_for_a/
43. Deploying ZFS Scratch Storage for NVMe on Azure Kubernetes ..., accessed May 19, 2025, https://techcommunity.microsoft.com/blog/azurehighperformancecomputingblog/deploying-zfs-scratch-storage-for-nvme-on-azure-kubernetes-service-aks/4363551
44. Managed Lustre | Google Cloud, accessed May 19, 2025, https://cloud.google.com/products/managed-lustre
45. IBM Storage Scale, accessed May 19, 2025, https://www.ibm.com/products/storage-scale
46. How to Build High-Performance NFS Storage with xiRAID Backend and RDMA Access, accessed May 19, 2025, https://xinnor.io/blog/how-to-build-high-performance-nfs-storage-with-xiraid-backend-and-rdma-access/
47. Silk AI Enablement:, accessed May 19, 2025, https://silk.us/wp-content/uploads/2025/01/Silk-AI-Enablement.pdf
48. Generally Available: Fully Managed Lustre File Storage in the Cloud - Oracle Blogs, accessed May 19, 2025, https://blogs.oracle.com/cloud-infrastructure/post/fully-managed-lustre-file-storage-in-the-cloud
49. Amazon FSx for Lustre now supports Elastic Fabric Adapter and NVIDIA GPUDirect Storage, accessed May 19, 2025, https://aws.amazon.com/about-aws/whats-new/2024/11/amazon-fsx-lustre-elastic-fabric-adapter-nvidia-gpudirect-storage/
50. Amazon FSx for Lustre increases throughput to GPU instances by up ..., accessed May 19, 2025,

https://aws.amazon.com/blogs/aws/amazon-fsx-for-lustre-unlocks-full-network-bandwidth-and-gpu-performance/
51. BeeGFS® at, accessed May 19, 2025, https://www.beegfs.io/docs/BeeGFS_UCSB_Whitepaper.pdf
52. AI & Deep Learning Solution Brief | ThinkparQ, accessed May 19, 2025, https://thinkparq.com/wp-content/uploads/2019/08/AI-Solution-Brief.pdf
53. IBM Storage Scale System, accessed May 19, 2025, https://www.ibm.com/products/storage-scale-system
54. GPUDirect Storage support for IBM Storage Scale, accessed May 19, 2025, https://www.ibm.com/docs/en/storage-scale/5.2.1?topic=architecture-gpudirect-storage-support-storage-scale
55. Installing GPUDirect Storage for IBM Storage Scale, accessed May 19, 2025, https://www.ibm.com/docs/en/storage-scale/5.2.1?topic=installing-gpudirect-storage-storage-scale
56. NVIDIA GPUDirect Storage: 4 Key Features, Ecosystem & Use Cases - Cloudian, accessed May 19, 2025, https://cloudian.com/guides/data-security/nvidia-gpudirect-storage-4-key-features-ecosystem-use-cases/
57. NVIDIA® GPUDirect® Storage - Digital Assets, accessed May 19, 2025, https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/tech-brief/tech-brief-nvidia-gpu-direct-openflex-data24-4000.pdf
58. Can I use NVIDIA's GPUDirect Storage to transfer data between GPUs and a parallel file system? - Massed Compute, accessed May 19, 2025, https://massedcompute.com/faq-answers/?question=Can+I+use+NVIDIA%27s+GPUDirect+Storage+to+transfer+data+between+GPUs+and+a+parallel+file+system%3F
59. Quantum unveils highly parallel file system client for Myriad - SDx Central, accessed May 19, 2025, https://www.sdxcentral.com/news/quantum-unveils-highly-parallel-file-system-client-for-myriad/
60. S3 Compatible Storage: Key Attributes, Benefits, Use Cases - Object First, accessed May 19, 2025, https://objectfirst.com/guides/data-storage/s3-compatible-storage-everything-you-need-to-know/
61. S3 Object Storage: The Ultimate Solution For AI/ML Data Lakes - StoneFly, Inc., accessed May 19, 2025, https://stonefly.com/blog/s3-object-storage-the-ultimate-solution-for-ai-ml-data-lakes/
62. Machine Learning Model Training with Alluxio: Part 2 - Comparable Analysis, accessed May 19, 2025, https://www.alluxio.io/blog/machine-learning-model-training-with-alluxio-part-2-comparable-analysis

63. Accelerating Data Loading in Large-Scale ML Training With Ray and Alluxio, accessed May 19, 2025, https://www.alluxio.io/blog/accelerating-data-loading-in-large-scale-ml-training-with-ray-and-alluxio
64. A Deep Dive into Caching in Presto - Alluxio, accessed May 19, 2025, https://www.alluxio.io/blog/a-deep-dive-into-caching-in-presto
65. A Journey Towards Data Locality on Cloud for Machine Learning and AI - Alluxio, accessed May 19, 2025, https://www.alluxio.io/blog/a-journey-towards-data-locality-on-cloud-for-machine-learning-and-ai
66. AI + Machine Learning File Storage Solution - JuiceFS, accessed May 19, 2025, https://juicefs.com/en/artificial-intelligence
67. JuiceFS - Open Source Distributed POSIX File System for Cloud, accessed May 19, 2025, https://juicefs.com/en/
68. DeepSeek 3FS vs. JuiceFS: Architectures, Features, and Innovations in AI Storage, accessed May 19, 2025, https://juicefs.com/en/blog/engineering/deepseek-3fs-vs-juicefs-architecture-feature
69. Flash Memory vs. RAM | Pure Storage Blog, accessed May 19, 2025, https://blog.purestorage.com/purely-educational/flash-memory-vs-ram/
70. SoftPerfect RAM Disk : high-performance RAM drive for Windows and macOS, accessed May 19, 2025, https://www.softperfect.com/products/ramdisk/
71. Backend Stores - MLflow, accessed May 19, 2025, https://mlflow.org/docs/latest/tracking/backend-stores
72. XFS vs. Ext4: Which Linux File System Is Better? | Pure Storage Blog, accessed May 19, 2025, https://blog.purestorage.com/purely-educational/xfs-vs-ext4-which-linux-file-system-is-better/
73. Understanding Linux File Systems: EXT4, XFS, BTRFS, and ZFS - Writeup DB, accessed May 19, 2025, https://www.writeup-db.com/understanding-linux-file-systems-ext4-xfs-btrfs-and-zfs/
74. What is Apache Spark ETL? Overview, Benefits & Use Cases - CData Software, accessed May 19, 2025, https://www.cdata.com/blog/what-is-apache-spark-etl
75. Ray vs Dask vs Apache Spark™ — Comparing Data Science & Machine Learning Engines, accessed May 19, 2025, https://www.onehouse.ai/blog/apache-spark-vs-ray-vs-dask-comparing-data-science-machine-learning-engines
76. accessed December 31, 1969, https://www.cdata.com/blog/what-is-apache-spark-etl/
77. Apache Spark + Kafka – Your Big Data Pipeline - Ksolves, accessed May 19, 2025, https://www.ksolves.com/blog/big-data/apache-spark-kafka-your-big-data-pipeline

78. Developing End-to-End Data Science Pipelines with Data Ingestion, Processing, and Visualization - KDnuggets, accessed May 19, 2025, https://www.kdnuggets.com/developing-end-to-end-data-science-pipelines-with-data-ingestion-processing-and-visualization
79. Generative AI Operations for Organizations with MLOps Investments - Azure Architecture Center | Microsoft Learn, accessed May 19, 2025, https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/genaiops-for-mlops
80. M29 - Distributed Data Loading - DTU-MLOps, accessed May 19, 2025, https://skaftenicki.github.io/dtu_mlops/s9_scalable_applications/data_loading/
81. Ray Data: Scalable Datasets for ML — Ray 2.46.0 - Ray Docs, accessed May 19, 2025, https://docs.ray.io/en/latest/data/data.html
82. Data Loading and Preprocessing — Ray 2.46.0 - Ray Docs, accessed May 19, 2025, https://docs.ray.io/en/latest/train/user-guides/data-loading-preprocessing.html
83. Preprocessing — dask-ml 2025.1.1 documentation, accessed May 19, 2025, https://ml.dask.org/preprocessing.html
84. Building End-to-End Data Pipelines with Dask - KDnuggets, accessed May 19, 2025, https://www.kdnuggets.com/building-end-to-end-data-pipelines-with-dask
85. Using Dask on Ray — Ray 2.46.0 - Ray Docs, accessed May 19, 2025, https://docs.ray.io/en/latest/ray-more-libs/dask-on-ray.html
86. cuFile API Reference Guide :: CUDA Toolkit Documentation - NVIDIA Docs Hub, accessed May 19, 2025, https://docs.nvidia.com/cuda/archive/11.7.1/cufile-api/index.html
87. Magnum IO GPUDirect Storage | NVIDIA Developer, accessed May 19, 2025, https://developer.nvidia.com/gpudirect-storage
88. NVIDIA GPUDirect Storage and MinIO AIStor: Unlocking Efficiency for GPU-Powered AI Workloads, accessed May 19, 2025, https://blog.min.io/nvidia-gpudirect-storage-and-aistor/
89. NVIDIA/gds-nvidia-fs: NVIDIA GPUDirect Storage Driver - GitHub, accessed May 19, 2025, https://github.com/NVIDIA/gds-nvidia-fs
90. accessed December 31, 1969, https://docs.nvidia.com/cuda/cufile-driver/index.html
91. Connectors that enable File-Based Storage | Storage Connectors for the PyTorch AI framework | Dell Technologies Info Hub, accessed May 19, 2025, https://infohub.delltechnologies.com/l/storage-connectors-for-the-pytorch-ai-framework/connectors-that-enable-file-based-storage/
92. Accelerate AI & Machine Learning Workflows | NVIDIA Run:ai, accessed May 19, 2025, https://www.run.ai/guides/ai-open-source-projects/nvidia-dali
93. rapidsai/kvikio: KvikIO - High Performance File IO - GitHub, accessed May 19, 2025, https://github.com/rapidsai/kvikio
94. Ecosystem | RAPIDS | RAPIDS | GPU Accelerated Data Science, accessed May 19,

2025, https://rapids.ai/ecosystem/
95. Feature Request: NVIDIA GDS Support for PyTorch IterableDataset & Checkpointing - data, accessed May 19, 2025, https://discuss.pytorch.org/t/feature-request-nvidia-gds-support-for-pytorch-iterabledataset-checkpointing/211945
96. Nvidia-dali-tf-plugin - General - Hailo Community, accessed May 19, 2025, https://community.hailo.ai/t/nvidia-dali-tf-plugin/12668
97. Memory management when using GPU in TensorFlow - Stack Overflow, accessed May 19, 2025, https://stackoverflow.com/questions/42307975/memory-management-when-using-gpu-in-tensorflow
98. TensorFlow Plugin API reference — NVIDIA DALI 1.9.0 documentation, accessed May 19, 2025, https://docs.nvidia.com/deeplearning/dali/archives/dali_190/user-guide/docs/plugins/tensorflow_plugin_api.html
99. What is ML Reproducibility - MLOps | MLOps Wiki - Censius AI, accessed May 19, 2025, https://censius.ai/wiki/ml-reproducibility
100. 13 ML Operations - Machine Learning Systems, accessed May 19, 2025, https://mlsysbook.ai/contents/core/ops/ops.html
101. 7.0. Reproducibility - MLOps Coding Course, accessed May 19, 2025, https://mlops-coding-course.fmind.dev/7.%20Observability/0.%20Reproducibility.html
102. MLOps Principles, accessed May 19, 2025, https://ml-ops.org/content/mlops-principles
103. Scalable AI Workflows: MLOps Tools Guide - Pronod Bharatiya's Blog, accessed May 19, 2025, https://data-intelligence.hashnode.dev/mlops-open-source-guide?source=more_articles_bottom_blogs
104. Reference Architecture: set up MLflow Remote Tracking Server on ..., accessed May 19, 2025, https://blog.ovhcloud.com/mlflow-remote-tracking-server-ovhcloud-databases-object-storage-ai-solutions/
105. Accelerate AI & Machine Learning Workflows | NVIDIA Run:ai, accessed May 19, 2025, https://www.run.ai/guides/kubernetes-architecture/kubeflow-pipelines-the-basics-and-a-quick-tutorial
106. Manage Kubeflow pipeline templates | Artifact Registry documentation - Google Cloud, accessed May 19, 2025, https://cloud.google.com/artifact-registry/docs/kfp
107. Pipeline Root | Kubeflow, accessed May 19, 2025, https://www.kubeflow.org/docs/components/pipelines/concepts/pipeline-root/
108. Object Store Configuration - Kubeflow, accessed May 19, 2025, https://www.kubeflow.org/docs/components/pipelines/operator-guides/configure

-object-store/
109. Remote Storage | Data Version Control · DVC, accessed May 19, 2025, https://dvc.org/doc/user-guide/data-management/remote-storage
110. Effortless Data and Model Versioning with DVC, accessed May 19, 2025, https://www.dasca.org/world-of-data-science/article/effortless-data-and-model-versioning-with-dvc
111. Versioning Data in MLOps with DVC (Data Version Control) - Full Stack Data Science, accessed May 19, 2025, https://fullstackdatascience.com/blogs/versioning-data-in-mlops-with-dvc-data-version-control-xm3mu5
112. Machine Learning Model Versioning: Top Tools & Best Practices, accessed May 19, 2025, https://lakefs.io/blog/model-versioning/
113. MLflow for gen AI agent and ML model lifecycle - Databricks Documentation, accessed May 19, 2025, https://docs.databricks.com/aws/en/mlflow/
114. MLflow Model Registry, accessed May 19, 2025, https://mlflow.org/docs/latest/model-registry/
115. Deploy MLflow Model as a Local Inference Server, accessed May 19, 2025, https://mlflow.org/docs/latest/deployment/deploy-model-locally/
116. DVC vs Git vs Git LFS: ML Reproducibility - Censius, accessed May 19, 2025, https://censius.ai/blogs/dvc-vs-git-and-git-lfs-in-machine-learning-reproducibility
117. Hyperparameter tuning - Databricks Documentation, accessed May 19, 2025, https://docs.databricks.com/gcp/en/machine-learning/automl-hyperparam-tuning
118. Hyperparameter tuning | Databricks Documentation, accessed May 19, 2025, https://docs.databricks.com/gcp/en/machine-learning/automl-hyperparam-tuning/
119. Ray Tune: Distributed Hyperparameter Optimization at Scale | RayTune-dcgan – Weights & Biases - Wandb, accessed May 19, 2025, https://wandb.ai/authors/RayTune-dcgan/reports/Ray-Tune-Distributed-Hyperparameter-Optimization-at-Scale--VmlldzoyMDEwNDY
120. Hyperparameter Tuning with Ray Tune — Ray 2.46.0 - Ray Docs, accessed May 19, 2025, https://docs.ray.io/en/latest/train/user-guides/hyperparameter-optimization.html
121. Running Distributed Experiments with Ray Tune — Ray 2.46.0 - Ray Docs, accessed May 19, 2025, https://docs.ray.io/en/latest/tune/tutorials/tune-distributed.html
122. Hyperparameter tuning with Optuna - Databricks Documentation, accessed May 19, 2025, https://docs.databricks.com/aws/en/machine-learning/automl-hyperparam-tuning/optuna
123. Distributed hyperparameter tuning with Optuna, Neon Postgres, and Kubernetes, accessed May 19, 2025, https://neon.tech/guides/optuna-hyperprameter-kubernetes

124. Machine Learning Operations Tools - Amazon SageMaker for MLOps - AWS, accessed May 19, 2025, https://aws.amazon.com/sagemaker-ai/mlops/