Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 3, Issue 1

Open Access | PP: 76-81

DOI: https://doi.org/10.70315/uloap.ulete.2026.0301013

Methods for Organizing Distributed Training of Large Language Models in Cloud Environments

Dhaval Shah

Senior Software Engineer, Redwood City, California, USA.

Abstract

Cloud environments make distributed LLM training a coupled systems problem: scaling forces partitioning, partitioning amplifies communication costs, and transient capacity elevates the cost of recovery. Coordinating model architecture choices with the communication stack and failure-handling policy is therefore necessary to preserve throughput under churn. In this setting, data, pipeline, and tensor parallelism are most effective when communication is overlapped with computation and checkpoint intervals are adapted to infrastructure behavior. This combination enables the use of low-cost unreliable instances without extending end-to-end training time, yielding resilient distributed training pipelines on elastic cloud platforms.

Keywords: *Large Language Models, Distributed Training, Cloud Computing, Hybrid Parallelism, Pipeline Parallelism, Communication Optimization, Fault Tolerance, Checkpointing, Kubernetes Orchestration, MLOps.*

INTRODUCTION

In practice, cloud-based training of large language models can become unstable for operational rather than strictly algorithmic reasons. Once the combined model state—parameters, optimizer buffers, and activations—no longer fits on a single GPU, the job is partitioned across several devices; that partitioning, in turn, tightens coupling through collective communication and synchronisation barriers. In pre-emptible settings the same coupling meets infrastructure churn: a virtual machine may be revoked, a pod rescheduled onto hardware with a different topology, or a placement rendered invalid as capacity fragments.

The bigger the model is the more taxing the scaling synchronization becomes for scaling. The necessity to achieve both scaling and speed of computing pushes the industry to decompose the model in different ways. It can be distributed across layers. Alternatively it is possible to copy it across GPUs. In the cloud, training usually runs inside containers that are launched on whatever machines are available at the time. There is often an assumption that training logic and infrastructure behavior can be treated separately. Yet when the compute environment is cloud-based, that separation breaks down. Nodes might not be available throughout the run; some are reclaimed, others spin up late. Decisions about when and where to persist state can govern the progress of training. The literature describes distribution techniques and fault-recovery mechanisms in detail, yet rarely examines them together. Consequently, the research lacks a clear picture of their combined behaviour when resources churn. This article seeks to fill that gap by analysing cloud-based LLM training as a single system and tracing how orchestration choices interact on elastic, failure-prone hardware.. This article closes that gap by analysing cloud-based LLM training as one end-to-end system and by mapping the interplay of its components on failure-prone hardware.

METHODS AND MATERIALS

Cho et al. [1] proposed ACUTE, a checkpointing method designed for use in spot-instance environments. Rather than writing checkpoints directly to disk, the system first stores them in a nearby on-demand VM's memory, reducing write delays and helping prevent loss during preemption. Decker and Kunkel [2] worked on a version of Kubernetes that lets high-performance computing nodes be reassigned quickly, even after failures. Guan et al. [3] explored how splitting models layer by layer across GPUs, along with other types of parallelism, helps reduce how much data has to move between devices. Kreuzberger et al. [4] looked at the bigger picture, connecting these sorts of system setups to the routines that teams follow when managing machine learning in practice. Lin et al. [5] introduced a system called OMStack built to handle a range of workloads.

Some of the literature explores Kubernetes scheduling. For

Citation: Dhaval Shah, "Methods for Organizing Distributed Training of Large Language Models in Cloud Environments", Universal Library of Engineering Technology, 2026; 3(1): 76-81. DOI: https://doi.org/10.70315/uloap.ulete.2026.0301013.

instance, Senjab et al. grouped schedulers by how they assign pods and juggle resources between containers [6]. Tran et al. took another approach, setting up a fault-tolerant system that ties into Kubernetes. They used Bi-LSTM models to watch time-series signals and guess when a node might fail—then moved services ahead of time to avoid any issues [7]. Weingram et al. [8], meanwhile, compared communication libraries like NCCL and Gloo, noting differences in gradient exchange overhead depending on the setup. Zeng et al. took a wider angle, laying out different training methods for large language models, including how data, model, pipeline, and even 3D parallelism can be used together [9]. Lastly, Zhang et al. walked through the types of compute platforms used—GPUs, TPUs, etc.—and covered memory handling, cluster interconnects, and distributed scheduling choices that affect overall performance [10].

Several papers focus on failure modes—some work on detecting them early, others on recovery once they occur [1][7]. Resource management shows up in a few places too, though often in terms of provisioning cycles or elasticity under load [2][5]. Communication between GPUs is a recurring concern [8], and there's scattered attention to workflow management and tooling practices [4]. Surveys usually go deep on architectures and scaling methods [9], whereas scheduling research sticks to things like node packing or admission rules [2][6]. Few of these efforts ask how model behavior shifts when training happens in unstable, cloud-based environments. Most treat the model and the infrastructure as separate problems, with little interaction between the two. What this paper tries to do is combine both—looking at how the structure of the model and the setup of the infrastructure affect each other during training.

RESULTS AND DISCUSSION

Most setups split the workload using data parallelism. Each GPU handles its own mini-batch, and gradients are averaged after the step. Another option is model parallelism—slicing up the model itself so that parts run on different devices. With large language models, though, one method isn't enough. They usually mix several: data, tensor (inside a layer), and pipeline (across layers). Guan et al. [3] describe pipeline parallelism as a way to split a model across layers, assigning different blocks to different GPUs. Micro-batches of data then move through those GPUs in sequence, stage by stage. It's not the only approach. Tensor parallelism goes inside a layer instead—dividing the math itself across devices—so multiple GPUs share the work within a single block. Figure 1 illustrates one such arrangement.

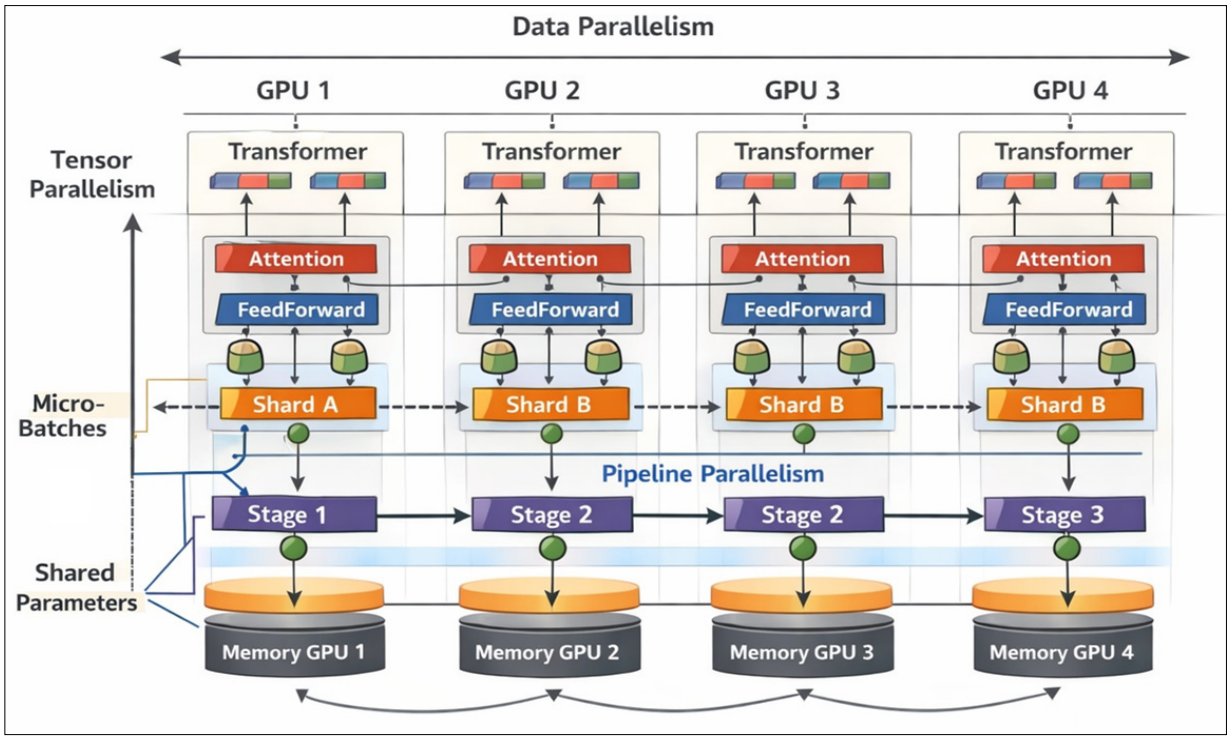

Figure 1. The Author's Illustration of the Hybrid (3D) Parallelism in Distributed LLM Training

At the top level of Figure 1, the same pipeline stages are copied across multiple GPUs or nodes. Each of them processes a different mini-batch, but they all work on the same model structure. After a while, the gradients from each replica have to be brought together and averaged so the model stays in sync. Below that tensor parallelism is shown. It goes inside a single layer and splits the operations across devices. So instead of each GPU running a full layer, they split the work within that layer itself. This setup spreads the memory load and also allows some of the layer's work to happen at the same time on multiple devices. It is mostly useful when a layer is too big to fit on one GPU. Pipeline parallelism is separate. It does not actually break the layers apart. It groups them—whole blocks—and maps them onto separate GPUs. Each block runs in its own phase, one after the other. Data, in the form of micro-batches, moves through the stages

sequentially. Since the batches are small and can move ahead on their own, different ones often end up at different places in the model at the same time.

As soon as training is spread across a large number of GPUs, compute is not the main issue anymore. The real bottleneck becomes communication—how fast gradients can be shared. Collective operations like AllReduce or Broadcast, or regular Send and Receive, happen in the middle of each step. If those are delayed, the whole iteration slows down [8]. For this reason, xCCL stacks do not treat collectives as auxiliary utilities. They provide several concrete implementations, most often ring- or tree-based and sometimes pipelined, and the choice among them depends on message size and the physical layout of the cluster rather than on a single fixed schedule [8]. One practical marker of the shift away from CPU-centric coordination is that GPU-oriented stacks prioritize GPU-resident reductions and direct NIC handoff: NCCL-type designs perform reductions on the GPUs and then rely on GPU-direct paths (for example, RDMA-capable fabrics such as InfiniBand or RoCE) so that inter-node transfers do not have to be marshaled through a CPU staging path [8].

Collective communication starts to differ across vendors and runtimes when heterogeneous clouds and mixed GPU setups are involved. On AMD systems using ROCm, RCCL keeps compatibility with the NCCL API but runs inside the ROCm stack, using HIP streams instead of CUDA. It works with local interconnects like PCIe and xGMI, and uses InfiniBand Verbs or TCP/IP sockets for communication between nodes. Profiling in this setup is handled through tools similar to NPKit [8]. The same survey also looks at setups where CPUs work together with accelerators. In those cases, oneCCL introduces a way to run collective operations in an asynchronous style. It uses communicators and events, and includes metadata for priorities, so some communication tasks can start early and keep going in the background while the main computation continues [8]. Figure 2 gives an example of this. It shows what happens during a single step in training, comparing a setup where communication waits until after computation with one where they happen at the same time.

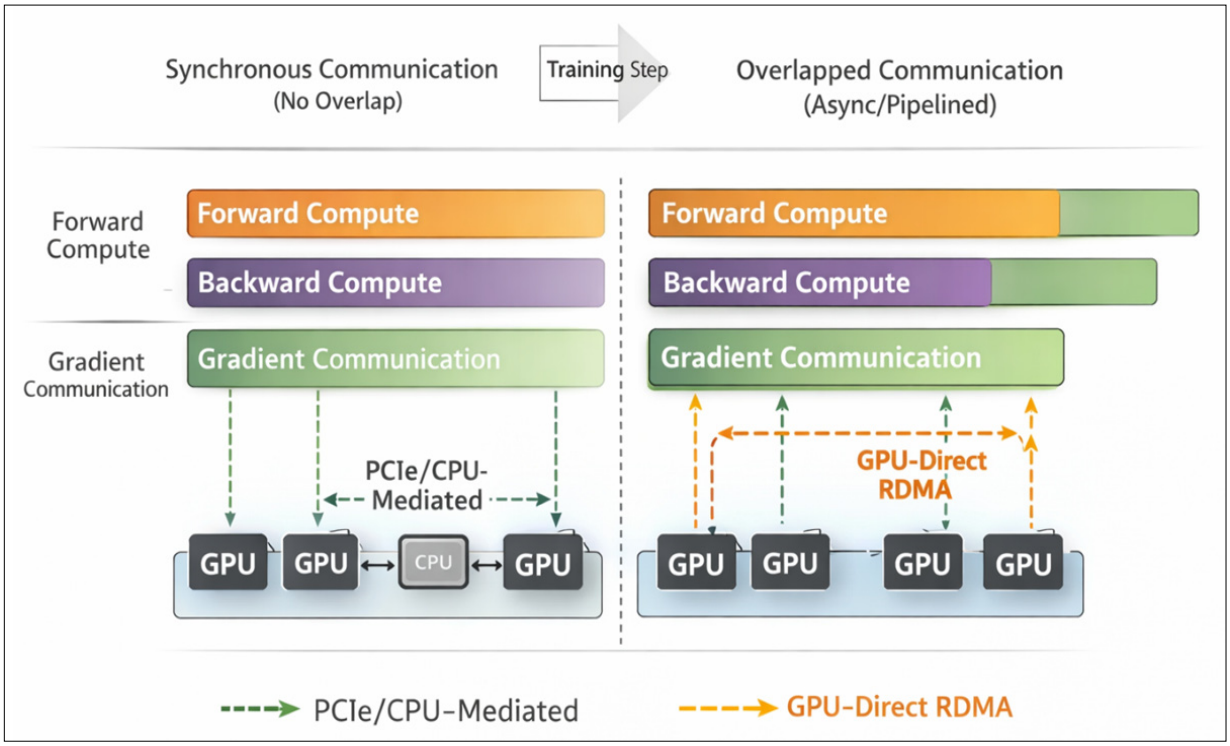

Figure 2. Communication–Computation Overlap and Optimized Gradient Exchange in Distributed LLM Training by the Author

On the left part of Figure 2, the process unfolds step by step: forward pass, then backward, then gradient transfer. When gradients have to move through the CPU using PCIe, the entire process can slow. The GPUs are not functioning for longer, which hurts scaling. That delay is one of the reasons why libraries like NCCL, RCCL, or oneCCL get used in real systems. They all handle similar kinds of collective operations, but they use different transport layers underneath. Some go over standard TCP/IP, others use InfiniBand Verbs, and a few rely on GPU-direct RDMA (GDR), which skips the CPU entirely when moving data. That last part can make a big difference, especially when trying to keep the GPUs busy without long stalls. The common implementations (ring and tree variants) make the overlap visible on a timeline: computation stays busy while communication drains in the background, rather than appearing as a hard synchronization wall. In that sense, the right side of the figure is not only overlapping communication with backward/forward compute; it is also implicitly assuming the RoCE/InfiniBand-class fabric needed to make GPU-to-GPU transfers the steady-state path (for example, the ZionEX configuration described in xCCL uses 8 GPUs each with a 200 Gbps RoCE NIC, and ACCL is discussed with 4× Mellanox ConnectX-5 100 Gbps NICs). Related pipeline markers also fit naturally into the same

concurrency framing: micro-batches are the unit that makes "many small concurrent operations" literal, and common schedules such as 1F1B and interleaved 1F1B are explicitly described as having almost no bubble overhead once the pipeline is filled.

Provisioning and cluster shape further determine whether those scheduling policies are even stable across runs. Lin et al. frame this as a service-stack problem: batch jobs require stateful runtime behaviors such as gang scheduling, and common add-ons like Kubeflow and Volcano are described as available but incomplete "adaptors," motivating a more universal abstraction for diverse scheduling modes [5]. The same source also gives a cloud-native "gotcha" that matters for LLM training images: CUDA and OFED versions can be coupled to host driver versions, so the stack has to decide what lives inside the container image versus outside it. Their implementation-level marker for elastic provisioning is OMStack, where OMCC treats a managed Kubernetes cluster as the first-class unit and supports on-demand provisioning, elastic scaling, and multi-cloud deployment; it also exposes an OMCC-OSB service catalog via the Open Service Broker interface. Decker and Kunkel push the "ephemeral cluster" idea to an extreme: Ephemeral Kubernetes (built on Warewulf images plus a HA mechanism using HAProxy, Keepalived, and a control-plane rejoin daemon called Phylactery) is reported as able to launch a five-node cluster in under 90 seconds and recover when nodes fail and rejoin . In this framing, "the cluster" becomes a disposable artifact rather than a persistent environment, which helps explain why long LLM runs often pair elasticity with fault-tolerance mechanisms.

Finally, that same cloud control plane becomes part of the recovery path, which is why "checkpointing discussed below" can be reinforced with a concrete Kubernetes-oriented marker rather than left generic. Tran et al. differentiate services whose QoS depends on booting state versus running state, and propose a proactive mechanism built around generalized snapshots and stateful migration. Reported effects include service completion time drops from 20 s to 12 s for a booting-state-dependent service (BSTD) and a 5 s improvement for a running-state-dependent service (RSDT), along with QoS-violation reductions of ~20% and ~22% respectively [9]. Figure 3 presents a simplified, layered view of how dist.

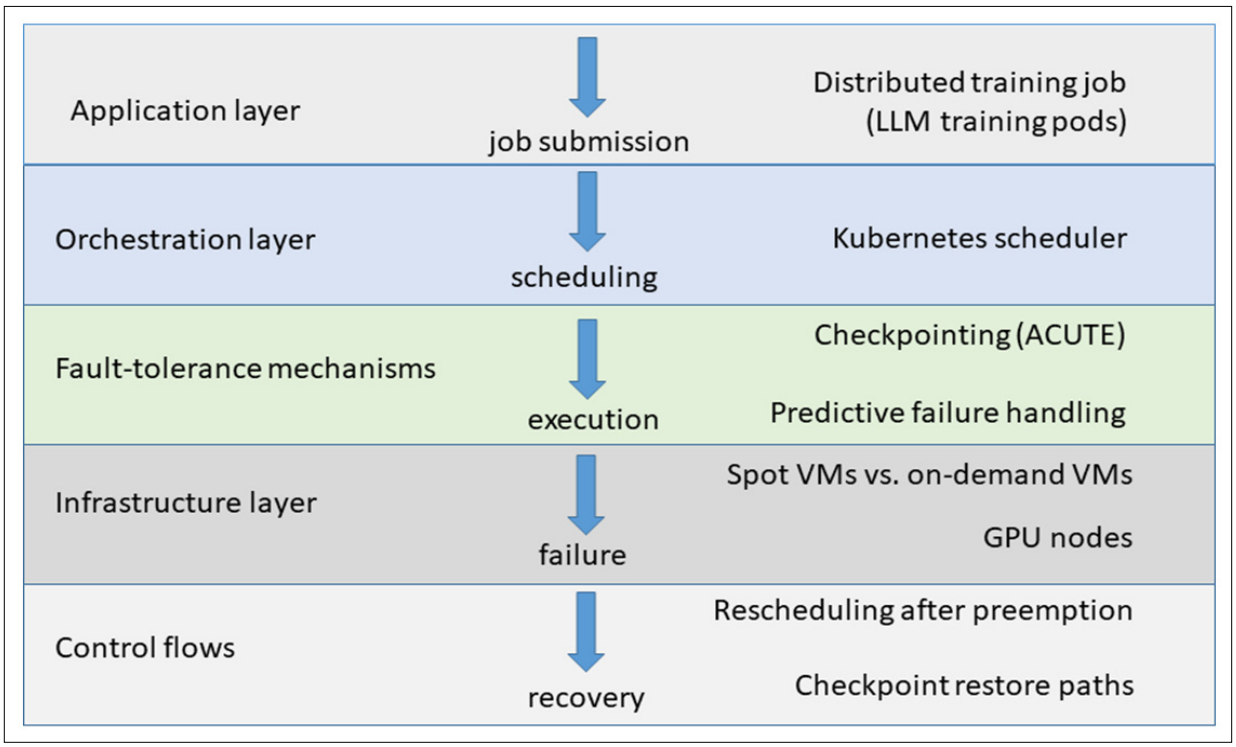

Figure 3. Cloud-Native Orchestration and Fault-Tolerant Execution of Distributed LLM Training

Figure 3 starts at the application level, showing up as a group of pods for training. Those pods then are passed to the orchestration layer, where Kubernetes ascertains where to place them. Training does not happen in isolation—checkpointing takes place alongside the computation, and there is also logic checking on the nodes while the job is going. On the infrastructure side, this usually runs on GPU nodes pulled from both on-demand and spot VMs. This is cost-efficient, but it also increases the chances of disruption, since spot instances might be reclaimed or a node might just fail. Subsequently, training is interrupted and the system switches to recovery. The affected pods get moved, the latest checkpoint gets loaded, and training resumes from there instead of restarting.

Cho et al. [1] describe a setup called ACUTE that is meant for spot VM clusters in the cloud. Instead of writing checkpoints immediately to shared storage, the method stages the model state in the memory of a nearby stable virtual machine, with persistent writes deferred. This arrangement does not change the semantics of recovery, but it alters how checkpoint

overhead appears during execution. ACUTE uses a two-step checkpoint path in which state is first staged in the memory of a nearby stable virtual machine and only later committed to persistent storage.

Tran et al. [7] propose a different solution based on failure prediction. Their system works by watching node-level signals and reacting when a failure seems likely, rather than after it has already happened. If the warning comes early enough, stateful containers are moved to another node before the crash. Keeping part of the state in memory helps limit how disruptive this is for training, although the benefit clearly depends on how reliable the predictions are. In practice this still means that checkpointing is needed. Cloud platforms may restart failed pods or replace preempted virtual machines automatically, but if there has not been a recent checkpoint, recovery is not feasible.

In reality, LLM training on the cloud does not happen as a single step. It transpires within a longer sequence—some parts before it, others after. Data have to be collected and cleaned first, parameters are tested repeatedly, results are checked, and only after that does deployment begin. For this reason, training is often embedded in broader workflows, not launched as an isolated job. Across the sources reviewed, efficiency appears to result less from any single technique and more from how well core elements—parallelism, communication, scheduling, and fault handling—are coordinated within the system that oversees the training process.

CONCLUSION

Cloud-based training of large language models rarely settles into a single standard execution pattern. Scaling changes stopped transferring between runs once the step time was dominated by synchronization rather than forward/backward kernels. Optimizer sharding and AMP reduced resident state per GPU, but the iteration boundary still collapsed onto gradient AllReduce in the xCCL layer. When that happened, the run's behavior depended less on arithmetic intensity and more on the collective path: a ring schedule that was acceptable on one topology became a hard stall on another, and the transport choice (InfiniBand Verbs with RDMA versus a socket path over TCP/IP) decided whether the tail stayed narrow or expanded into a full barrier [8][9]. GPU-direct paths helped only when the fabric and placement matched the assumed topology; otherwise the same job spent the step budget waiting at the collective.

Cloud-native placement surfaced a related constraint before steady-state training existed at all. Large contiguous GPU requests and gang-style allocation across the replica set delayed admission, and pods could sit unbound while the scheduler searched for eligible nodes that satisfied the GPU, topology, and quota constraints [9]. Restart behavior varied with checkpoint layout. When model weights and optimizer state were split across many small objects, recovery blocked during enumeration and fetch. Training did not resume until the final shard arrived; earlier shards did not help. In several cases, most of the state was already present, but the job still waited on metadata registration before the runtime could proceed. Kubernetes-based recovery shortened the outage by restoring from an existing snapshot after node loss or pod eviction. Meanwhile, at the infrastructure layer, ephemeral cluster lifecycles (for example, deleting and recreating Kubernetes clusters via Warewulf-based workflows) reinforce the same point: "the cluster" is often a transient artifact, so training systems have to treat rescheduling and rebuild as normal operations, not exceptional events [2].

Finally, once training becomes a repeatable organizational capability rather than a one-off experiment, the paper's findings converge on an operational claim: scaling and reliability only remain stable when MLOps-style workflow structure is present from the beginning, not bolted on after the first failures. Memory relief (mixed precision, ZeRO-style optimizer or parameter partitioning) can move the limit to collective latency in All-Reduce; changes that reduce collective overhead (GPU-direct RDMA, topology-matched collectives) can make scheduler queuing, GPU fragmentation, or checkpoint restore time dominate instead [8][6][10].

Future work can make this coupling measurable under realistic cloud volatility by replaying spot/preemption traces and reporting step-level and run-level outcomes (makespan, lost work, recovery time, and storage traffic) while sweeping ACUTE-style multi-level checkpoint hierarchies (node-local staging plus shared object storage) and checkpoint interval policies; the target is to preserve near-zero makespan impact (<0.01%) while retaining reported checkpoint-time speedups (~43.30%) and pause-time reductions (~53.1%) at checkpoint sizes on the order of 667 MB–1.31 GB [1]. Scheduler effects are isolated by keeping the hybrid decomposition constant (pipeline stages, tensor/model shards, data replicas) and changing only pod placement and admission behavior. The same layout can be run under Volcano-style gang scheduling, topology-aware placement, and preemption-aware backfilling. Measurement separates collective wait time (GPU idle attributed to pending collectives) from straggler amplification (step-time stretch induced by the slowest replica). Scheduling and recovery policies can therefore be compared under identical model partitioning and communication settings.

REFERENCES

1. Cho, Y., Kim, Y., Kim, K., Kim, J., Kim, H.-Y., & Kim, Y. (2024). Optimizing multi-level checkpointing for distributed deep learning workloads on cloud spot VM clusters. *IEEE Access*, *12*, 116891–116904. https://doi.org/10.1109/ACCESS.2024.3446770

2. Decker, J. & Kunkel, J. Ephemeral Kubernetes: dynamically deleting and recreating clusters using Warewulf. *The Journal of Supercomputing*, *81*(1491), 1-28. https://doi.org/10.1007/s11227-025-07668-y

3. Guan, L., Li, D.-S., Liang, J.-Y., Wang, W.-J., Ge, K.-S., & Lu, X.-C. (2024). Advances of Pipeline Model Parallelism for Deep Learning Training: An Overview. *Journal of Computer Science and Technology*, *39*(3), 567–584. https://doi.org/10.1007/s11390-024-3872-3

4. Kreuzberger, D., Kühl, N., & Hirschl, S. (2023). Machine learning operations (MLOps): Overview, definition, and architecture. *IEEE Access*, *11*, 31866–31879. https://doi.org/10.1109/ACCESS.2023.3262138

5. Lin, J., Xie, D., Huang, J., Liao, Z., & Ye, L. (2022). A multi-dimensional extensible cloud-native service stack for enterprises. *Journal of Cloud Computing: Advances, Systems and Applications*, *11*(70), 1-18. https://doi.org/10.1186/s13677-022-00366-7

6. Senjab, K., Abbas, S., Ahmed, N., & Khan, A. u. R. (2023). A survey of Kubernetes scheduling algorithms. *Journal of Cloud Computing: Advances, Systems and Applications*, *12*, (69), 1-26. https://doi.org/10.1186/s13677-023-00471-1

7. Tran, M., Vo, B., Pham, A., Pham, T.-L., & Nguyen, K.-T. (2022). Proactive stateful fault-tolerant system for Kubernetes-based containerized services. *IEEE Access*, *10*, 102181–102194. https://doi.org/10.1109/ACCESS.2022.3209257

8. Weingram, A., Li, Y., Qi, H., Ng, D., Dai, L., & Lu, X. (2023). xCCL: A survey of industry-led collective communication libraries for deep learning. *Journal of Computer Science and Technology*, *38*(1), 166–195. https://doi.org/10.1007/s11390-023-2894-6

9. Zeng, F., Gan, W., Wang, Y., & Yu, P. S. (2025). Distributed training of large language models: A survey. *Natural Language Processing Journal*, *12*(100174), 1-20. https://doi.org/10.1016/j.nlp.2025.100174

10. Zhang, Z.-X., Wen, Y.-B., Lyu, H.-Q., Liu, C., Zhang, R., Li, X.-Q., Wang, C., Du, Z.-D., Guo, Q., Li, L., Zhou, X.-H., & Chen, Y.-J. (2025). AI computing systems for large language models training. *Journal of Computer Science and Technology*, *40*(1), 6–41. https://doi.org/10.1007/s11390-024-4178-1