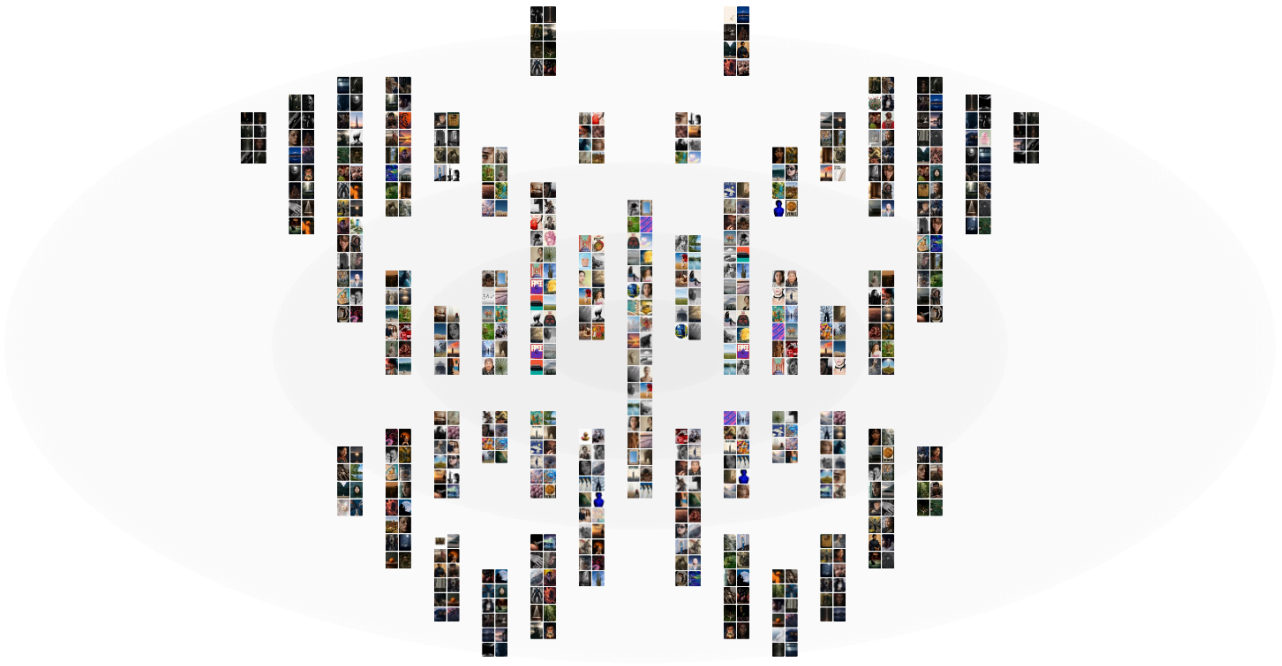


Training Reve 2.0 at scale

A technical report on the infrastructure behind our multimodal model



Reve Research Team

Report by Joe Wang with contributions from Kyra Mo and Ben Cuan

When the Reve Research Team formed in 2024, we set out to reimagine visual intelligence in creative tooling. We researched new architectures and trained multimodal models to expand the control and expression possible in generative images. We were guided by one principle: creative intent over technical proficiency.

In just two years, we invented a novel layout representation, scaled our infrastructure to handle training at ten-thousand-GPU scale, and brought direct spatial, semantic, and color control to image generation and editing. Reve 2.0 and 2.1 achieved results unparalleled by a sub-\$1T company, trained on 10x fewer GPUs.

Our constraints on engineering time and compute, combined with the need to support diverse input and output formats across modalities, shaped the development of our infrastructure. The result is a training platform built around a simple, unified interface for researchers, efficient resource allocation, cross-region scheduling, and automated fault recovery.

We're releasing this report as a reference for the lessons and techniques gathered while training Reve 2.0, 2.1, and their predecessors. In the following sections, we cover our compute platform and job management tool, explaining how we place and run jobs across regions, contain failures automatically, and diagnose performance. If you're interested in reference source code, email us at training-infra@reve.art to let us know what you'd find useful.

01 System Overview

We use Kubernetes [1] as our compute platform, and Ray [2] for distributed training. Our infrastructure runs on Google Cloud Platform[3] (GCP).

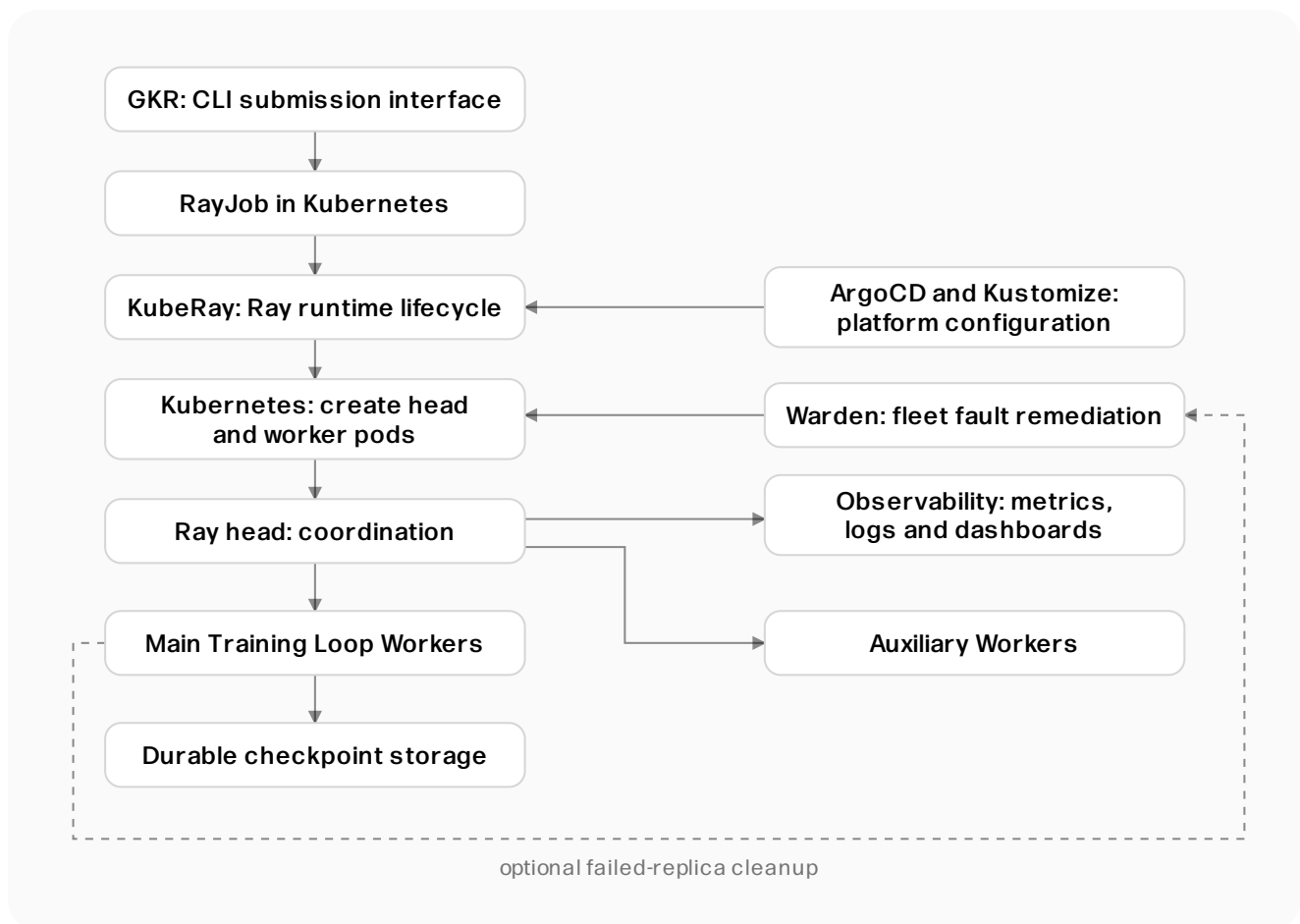


Figure 1: Compute platform architecture

We created an in-house workload client, which we internally refer to as GKR (short for Google Kubernetes Engine Runner). KubeRay [4] manages the Ray job lifecycle; PyTorch [5] and Ray Train [6] facilitate distributed computation; and Warden detects and resolves node faults. The underlying cluster and platform are configured through a GitOps system built with ArgoCD [7], Terraform [8], and Kustomize [9].

02 Workload Interface and Submission Flow

2.1 Unified CLI tool

GKR is an internal tool used by our researchers to interact with our infrastructure. It is a cluster-aware command-line interface that supports starting development shells, running and managing jobs, inspecting resources and logs, accessing services, transferring files, and handling many other day-to-day workflows across our compute platform. Keeping all the infrastructure tooling under a single CLI gives researchers a consistent entry point and reduces the number of systems they need to learn.

While GKR is built around GCP's ecosystem, core functions like submission, inspection, and resource management apply to any Kubernetes-based compute infrastructure.

2.2 Job configuration

For submitting jobs, GKR accepts command-line arguments or YAML files. It resolves default values and generates the Kubernetes resources through Jinja [10] templates. The interface hides platform- and hardware-specific boilerplate from the researchers.

For example, the following request configures the compute layout, resources, runtime environment, and scheduling of a training job:

```
gkr submit TRAINING_JOB rayjob \  
  --cluster TARGET \  
  --namespace TEAM \  
  --pods 2 \  
  --gpus 8 \  
  --cpus 80 \  
  --memory 512Gi \  
  --accelerators H200 \  
  --docker-image IMAGE \  
  --working-dir /PATH/TO/REPO \  
  --env /PATH/TO/ENV.yaml \  
  --preemptible \  
  --cmd 'uv run python train.py path/to/config.yaml'
```

GPU, CPU, and memory requests are per pod. This request defines the job's overall resource allocation. GKR then translates these requirements into Kubernetes resource requests, selectors, tolerations, and affinity rules in the manifests it generates.

GKR has three main submission types. A RayJob runs training in a temporary RayCluster. A persistent RayCluster is reusable and requires explicit deletion. A Job supports work that does not need Ray, commonly used for temporary supplementary services or development pods.

2.3 Application configuration

Researchers configure the job's training application through a standardized training driver and a YAML file. The file describes training parameters, data input, compute layout, logging, algorithms, and other settings. This separates application configuration from the Kubernetes launch templates.

We use the Hydra [11] library to combine reusable defaults with the selected YAML file and command-line overrides:

```
python train.py path/to/config.yaml run.name=example
```

This design allows researchers to express training patterns as YAML fields, making it easy to swap components and tweak run parameters across experiments. `train.py` is the training driver entrypoint.

2.4 Validation and preflight checks

Before a long-running job consumes significant GPU time, GKR performs extensive checks for faulty nodes and invalid configurations. GKR checks names, argument combinations, and rendered Kubernetes manifests before `kubectl apply`. After the resources are applied, a GKR preflight examines the pods that the cluster has placed. The preflight reads kernel log entries and command outputs to check for memory, hardware, and driver errors. It also tests computation and collective operations, and verifies shared mounts, libraries, and transport settings. When an error is encountered, GKR offers the job owner the option to tear down the `RayCluster`.

03 Platform Deployment

3.1 Reservations and hardware contracts

We use capacity from Google Cloud Platform future reservations. Compared to regular spot instances, future reservations guarantee us a reliable and colocated block over a longer time horizon [12], which is essential for running uninterrupted thousand-GPU training runs. Therefore, we built our infrastructure under the assumption of a relatively long-term hardware reservation. Our cluster uses managed Kubernetes control planes through Google Kubernetes Engine [13] (GKE), and the primary training node families are eight-GPU A3 Mega and A3 Ultra [14]:

Machine type	GPUs	Host CPU / memory	Local SSD	Network	Max network bandwidth
a3-megagpu-8g	8 × H100, 80 GB each	208 vCPU / 1,872 GB	6,000 GiB	GPUDirect-TCPXO	1,800 Gbps
a3-ultragpu-8g	8 × H200, 141 GB each	224 vCPU / 2,952 GB	12,000 GiB	RoCE with gIB plugin	3,600 Gbps

Table 1: Training node families

During job submission, GKR resolves the machine type from the accelerator type and handles the network differences between GPU models by specifying the correct interface, library mounts, and NCCL [15] tuning variables in pod configurations. Our platform does not support heterogeneous accelerators in a single job, though researchers can use the cross-region implementation described in section 6 to submit multiple decoupled learners with different accelerators as part of the same training experiment.

3.2 Provisioning and image delivery

Git tracks the platform's configuration. Terraform provisions cloud resources, and ArgoCD reconciles resources against manifests composed with Kustomize. Training images are built by CI, pinned to a digest in GKR, and cached asynchronously on each node to reduce job start time.

04 Cluster-level Job Orchestration

4.1 Job admission and placement

Precise resource placement is essential for supporting performance engineering. [16]

On job submission, GKR applies the `RayJob` or `RayCluster` manifests to Kubernetes. Kueue [17] is configured to admit all jobs regardless of resource availability. The KubeRay operator watches these manifests and creates Ray pods. The Kubernetes scheduler then places the pods, which form a `RayCluster`.

During placement, the scheduler matches the requirements from the GKR-generated pod specifications to available cluster nodes. For example, we use selectors to separate dedicated Ray head CPU nodes from accelerator nodes, and use tolerations to enforce accelerator type and other internal resource group constraints.

Furthermore, whole-node, eight-GPU allocation always preserves intra-node GPU locality. GKR also supports rack affinity by using GKE topology labels. However, internal empirical testing showed that rack affinity has minimal impact on our training workloads within the same GCP region. We expect this feature to be more beneficial for workloads running on GCP Vera Rubin A5X machines, since each rack would form a single 72-GPU NVLink domain [18].

Our platform has no capacity-based workload queueing or guaranteed whole-job readiness. Pods that cannot be handled by current capacity remain pending and are scheduled once resources become available. We decided not to implement a queueing system because we preferred to resolve resource conflicts through manual coordination outside the platform. For teams that require automated queueing, configuring capacity limits and admission policies in Kueue [17] would be the natural next step.

4.2 Job preemptibility

Some tasks like data curation can continue running after losing worker pods, while most training jobs are tuned to or depend on a specific training world size. To improve cluster utilization, jobs that can tolerate worker loss can be marked as preemptible through a GKR flag. These jobs are usually submitted with lower pod priority, allowing normal, non-preemptible jobs to reclaim resources from them. GKR assigns:

GKR Job Priority	Pod priority class
Normal, non-preemptible	default-priority
--preemptible	batch-gpu-preemptible-priority

For example, Kubernetes may terminate a preemptible pod to allow the creation of a pending higher-priority pod. KubeRay then creates a replacement pod, which remains pending until resources become available. The Ray worker loss would be handled by the Ray head.

4.3 Job lifetime

For `RayJob`, KubeRay enforces `shutdownAfterJobFinishes`, so the `RayCluster` is torn down when the training command terminates, whether it succeeds or fails. A `RayJob` can be configured to temporarily reserve its former nodes for the same namespace after its worker pods are deleted, making relaunch easier. For a `RayCluster`, GKR creates a long-lived sentinel pod that points to a persistent `RayCluster`. That cluster can process multiple jobs sequentially without recreating the Ray runtime and must be explicitly deleted by its owner. `RayClusters` are commonly used by researchers as a way to reserve capacity on the cluster.

05 Distributed Execution in Ray

5.1 RayCluster coordination

A `RayCluster` is formed by one Ray head pod [19] and many Ray worker pods. The Ray head is responsible for coordinating `RayCluster` membership, resources, scheduling, and control data. Its Global Control Service tracks cluster state and supports core Ray functions like placement groups, actors, and job scheduling. After KubeRay starts a `RayCluster`, the Raylets on its worker pods register their resources with the Ray head. The `RayCluster` cannot use unplaced pods or free GPUs from another `RayCluster`.

The training driver introduced in section 2.3 runs on the Ray head pod. The driver plans worker capacity based on workload specifications and coordinates training processes using Ray's APIs. The driver also uses Ray Train, Ray's distributed-training library, to initialize and handle the training workers, each implemented as a Ray actor. Finally, it waits for the job to complete and handles terminal failures.

Before discussing capacity planning, we need to introduce auxiliary workers.

5.2 Auxiliary workers

Alongside the **training workers**, which execute the main training loop, a job may also include other services like inference, encoding, and evaluation, each with vastly different timescales and resource requirements. Our **auxiliary workers** also implement these services as persistent Ray actors. Each service can have its own startup checks, failure handling, and timeouts.

Resource pools for auxiliary workers are allocated separately from those for training workers, and each auxiliary service may have several worker replicas behind a shared proxy for that service on each node. The only exceptions are singleton auxiliary services whose implementations cannot be safely distributed across replicas. Resource allocation is discussed further in [Section 5.3](#).

This design treats auxiliary tasks as microservices within the `RayCluster`, allowing better resource utilization and features like load balancing and queueing. Keeping them within the same Ray job also saves researchers from managing multiple separate Kubernetes deployments.

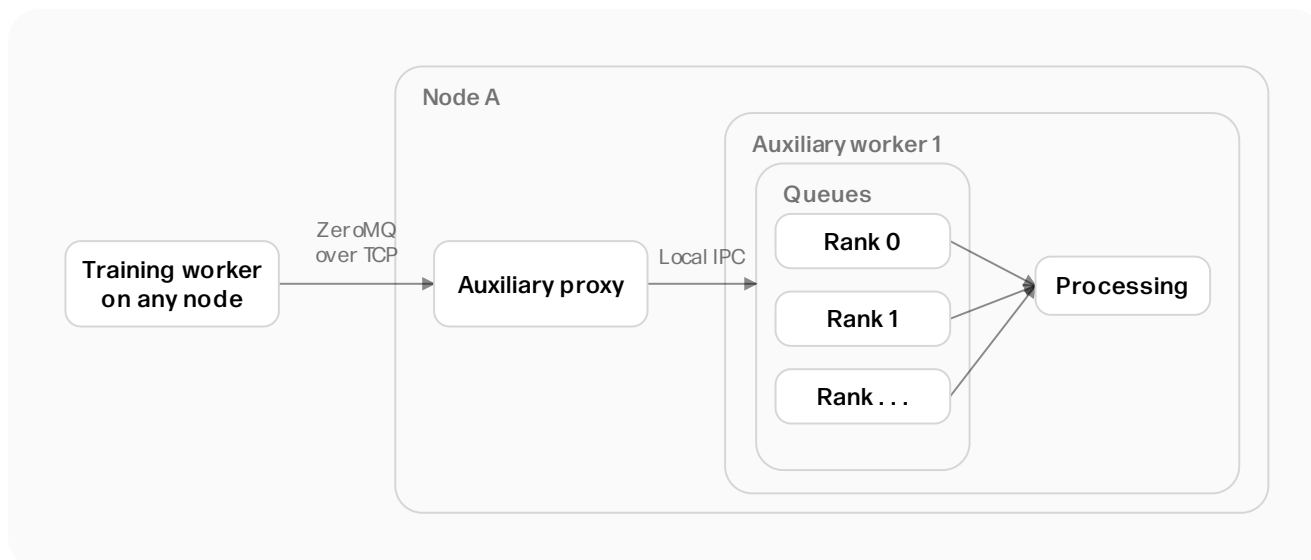


Figure 2: Auxiliary service architecture

The communication between a proxy and its auxiliary workers occurs through long-lived interprocess sockets, while messages between training workers and proxies go through ZeroMQ [20] sockets over TCP—even when they are on the same node. Earlier implementations used Ray RPC for communication between training workers and proxies, but we later adopted ZeroMQ with a custom remote-call layer for a substantial performance improvement. Weight transfers are the exception and use custom implementations of direct NCCL refreshes between training and auxiliary workers. Figure 2 illustrates this auxiliary service architecture.

For communication over ZeroMQ, the training workers distribute each type of auxiliary-worker request using weighted round-robin, with each auxiliary proxy's weight proportional to its worker count and no spatial affinity. Auxiliary proxies forward requests to their workers in round-robin order. Each worker maintains one internal bounded queue for each training rank and dequeues requests from these queues in round-robin order. This way, we prevent a few busy ranks from monopolizing request admission.

5.3 RayCluster capacity planning

Effective initial planning is essential for efficient resource utilization. [21] The training driver performs the capacity planning by dividing available GPUs and CPUs between training and auxiliary workers. It decides how many replicas each service type should have.

The driver reserves capacity for auxiliary services before training workers. It makes this decision using a proportional estimate based on each worker's resource request and the configured ratio of training workers served per auxiliary worker. Each training worker should occupy one whole GPU, but several auxiliary workers can share a single accelerator. Note that a single node can host any type of worker. We calculate the estimated number of training workers with:

G = total registered GPUs,
 C = total registered CPUs,
 g_i = GPUs per auxiliary worker of type i ,
 c_i = CPUs per auxiliary worker of type i ,
 m_i = trainer processes served per worker of type i ,
 c_t = CPUs requested per trainer process.

$$N_{\text{train}} = \min \left(\frac{G}{1 + \sum_i \frac{g_i}{m_i}}, \frac{C}{c_t + \sum_i \frac{c_i}{m_i}} \right).$$

The formula provides a proportional estimate. The driver then applies other constraints, like rounding and fulfilling minimum per-service allocations. For example, we allow multiple types of auxiliary workers to share an accelerator during single-node debugging runs, but enforce a minimum of one GPU per service type on multi-node jobs to ensure predictable placement and performance. For training worker allocation, the driver also rounds the number based on the parallelism layout [22]. The result specifies the world size of the main training loop and the worker allocation for each type of auxiliary service.

Engineers can use auxiliary worker metrics from the FleetMonitor discussed in section 8.2 to tweak each worker's resource allocation and the training-to-auxiliary ratio.

5.4 RayCluster worker placement

An auxiliary worker coordinator running on rank 0 transforms the auxiliary-worker resource assignments into Ray placement groups [23] and creates the workers and service proxies. The default policy for auxiliary workers is SPREAD, which spreads resource bundles across cluster nodes on a best-effort basis to improve fault isolation.

Ray Train, running on the training driver, initializes the PyTorch Distributed [24] process group based on the number of accelerators allocated to the main training loop. The driver launches one NUMA-affine Ray worker process per training GPU, with each process serving as a distributed rank. A Ray actor is initialized within each process and becomes the training worker. It receives the training configurations and executes the distributed program using PyTorch and NCCL. Figure 3 illustrates the creation of training workers.

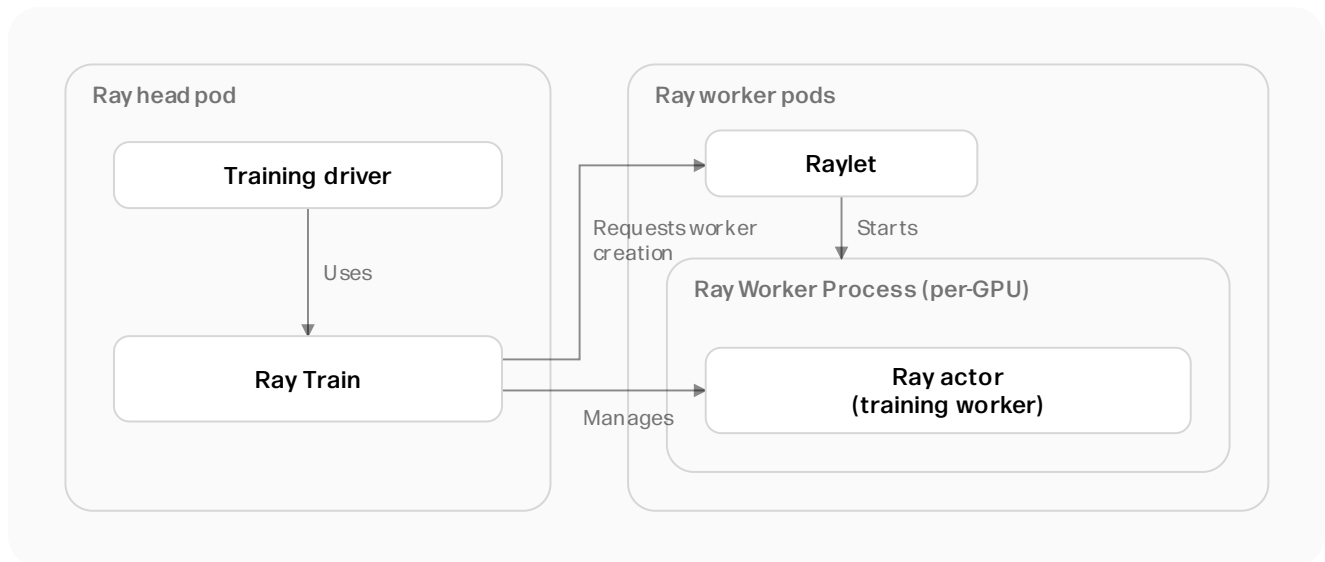


Figure 3: Ray Train worker creation

5.5 Training data pipeline

Fast data ingestion is essential for high-performance training jobs.

In the standard configuration, each training rank owns a data loader pipeline. Data loaders were initially implemented as Ray actors, but because each rank required several parallel loaders, this approach created tens of thousands of actors and put significant strain on Ray's Global Control Service. The latest implementation uses multiprocessing backends that are decoupled from Ray. This decoupling is one reason we switched from `ray.get` to ZeroMQ for communication with auxiliary workers.

If deterministic sampling is enabled, each data loader backend computes the same global sampling schedule and derives its sample slice from its rank. The deterministic loader also records sampling progress so that training can resume from a checkpoint without restarting the sampling schedule.

In each data loader pipeline, a feeder process queues sample indices, parallel producer processes fetch and transform the samples, and parallel collator processes turn them into batches for the training process. Each stage uses shared memory and bounded queues [25] to transfer data. The stages can overlap with one another, creating a steady batch stream for the training workers.

Select stages of the data loader pipeline can also run on auxiliary workers rather than alongside each training rank. In those cases, each auxiliary data loader would serve multiple training workers.

06 Cross-region Jobs

We organize our training clusters by GCP region. Due to capacity constraints, some training jobs had to span multiple regions, where network latency and bandwidth become much more significant bottlenecks. Over time, multi-region support grew from simply linking experiments into a dedicated training path involving a custom variant of Distributed Low-Communication training (DiLoCo) [26].

6.1 Local learners

Similar to canonical DiLoCo, we split the training job into independent local learners, each running in its own `RayCluster` with its own resource allocation and lifecycle. Each learner controls its own model replicas and optimizer state, and most training steps run locally within each learner.

After many local updates, a central synchronization pod requests weight snapshots from all participating learners. The synchronizer merges the snapshots, applies a separate outer optimizer to update the shared model, and broadcasts the weights back. The merging rules and outer optimizer algorithms can vary based on implementation.

The learners are `RayJobs` that share a single logical run identity. Cross-cluster messages and weights are transmitted using ZeroMQ over TCP, and intra-cluster GPU communication is handled by NCCL.

Figure 4 illustrates this DiLoCo architecture.

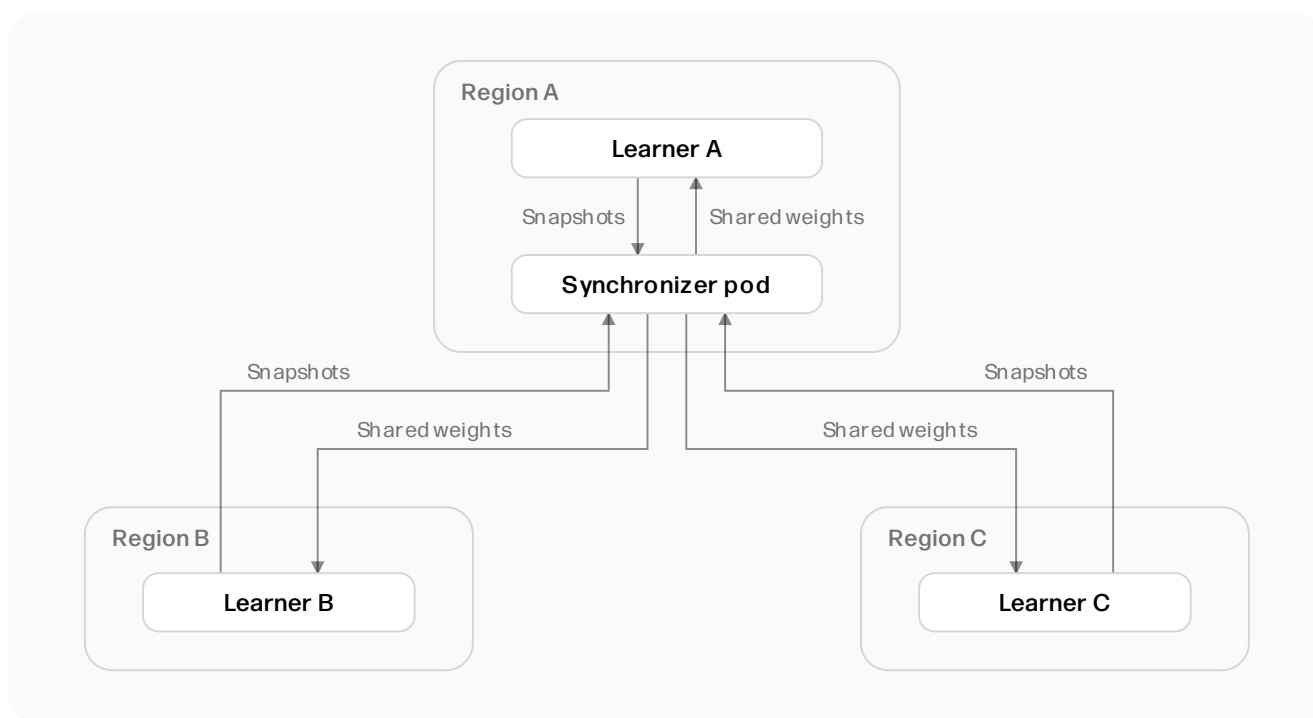


Figure 4: DiLoCo architecture

6.2 Barrier and decoupled modes

We implement two synchronization modes: a barrier mode, in which local learners block until they receive the weight merges; and a decoupled mode [27], in which learners continue training locally and merge the changes once the shared update is available. Because local training may have advanced beyond the shared update, decoupled mode requires researchers to configure additional rules for merging or replacing the local weights.

07 Failure Containment and Recovery

Failure handling is one of the core challenges of running any large-scale GPU platform. [28] [29] Our early implementations focused on failure detection and node cordoning, while later work expanded the capabilities to include autonomous recovery and elastic scale-downs.

7.1 Checkpointing

Checkpointing is the fundamental mechanism for preserving training progress across job failures.

We use PyTorch Distributed Checkpoint [30] (DCP) for saving and restoring interrupted runs. When checkpointing is triggered after a step, the lowest-index replica group takes a snapshot of its training state and saves it in CPU memory. Each rank stores its shard independently; DCP handles metadata indexing and mapping between tensors and stored shards. In addition to saving its own shard, the lowest-index rank is also responsible for triggering callbacks for auxiliary worker checkpointing. The main training loop continues after the snapshot is staged on host memory, and a background DCP thread writes the CPU copy to Google Cloud Storage.

Ray Train may recover from a worker-group failure by restarting the training group from the last checkpoint, or a researcher can submit a GKR job with an explicit resume checkpoint. On resume, we provide the desired distributed layout while DCP aligns and loads sharded training states into the training processes. The optional torchft integration discussed in the following subsection offers an alternative recovery path.

7.2 Ray auto-shrink and torchft fault tolerance

On the default failure recovery path, all progress made after the last checkpoint is lost when a single worker fails. At a hypothetical scale of 20,000 GPUs, our estimated mean time between failures (MTBF) would be shorter than our checkpoint interval, so implementing live fault tolerance is non-negotiable.

The optional torchft [31] integration allows the `RayCluster` to auto-shrink and keep the surviving model replica groups (e.g., FSDP [32] groups) running even after the loss of a trainer process.

The Ray driver hosts a torchft Lighthouse, which tracks training replica groups and maintains the quorum. The lowest rank of each training replica group runs a manager server that sends heartbeats to the Lighthouse. On each training step, the manager servers enforce a distributed, all-or-nothing optimizer update within their replica groups.

Liveness watchdogs in replica groups maintain membership across shards and tear down the entire group when a shard fails. The cluster Lighthouse detects missing heartbeats and reforms the replica-group quorum. If at least the configured minimum number of replicas survive, the groups rebuild cross-replica gradient communication and the training continues. If not, the training terminates.

When a replica group is dropped, the fault-tolerance controller on the lowest-index rank requests cleanup from the cluster health worker (Warden). Warden shrinks the `RayCluster` definition; KubeRay then reconciles the `RayCluster` with that smaller allocation, all without interrupting training. The next section describes Warden's role in more detail.

7.3 Automatic node repair with Warden

At our scale, hardware failures are routine. We built Warden as a fleet health manager that requires minimal human intervention.

Cluster repairs are implemented as periodic Temporal [33] workflows executed by Warden pods. Each node has a unique workflow, and each workflow's identity and progress are stored in the Temporal database. This setup ensures that only one worker can operate on a node at a time and that remediation sequences are preserved across worker restarts.

The detection workflow runs periodically; it queries GPU telemetry and inspects Kubernetes state to probe for hardware or driver errors. A definite fault immediately cordons the node, while ambiguous faults trigger `gkx check` for more comprehensive testing. The check starts a Kubernetes job that runs the preflight test suite described in section 2.4.

A periodic recovery workflow sweeps cordoned nodes and reboots them when there are no user workloads. It can also drain preemptible single-node jobs after notifying their owners and allowing a grace period. After the reboot, the stress-test workflow runs GPU diagnostics [34] and uncordons the node if the diagnostics pass. Figure 5 illustrates this fleet-remediation process.

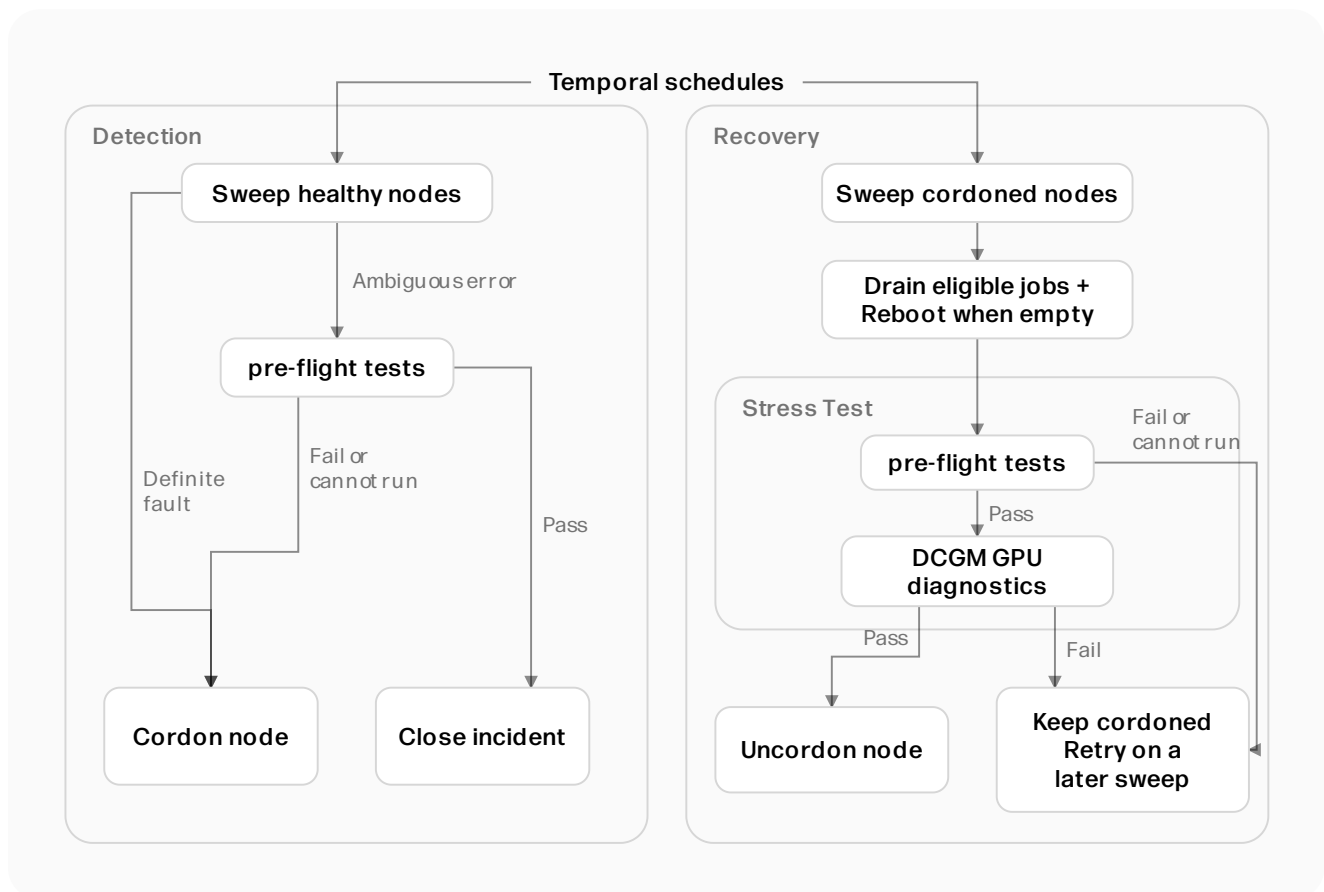


Figure 5: Warden fleet remediation

When cordoning a node, in addition to the node-spec modification made by `kubectl cordon`, Warden uses custom annotations to retain information across incidents, such as the number of recent cordons and the time of the last cordon. Warden can then use these annotations to guide future recovery or escalation decisions. For example, repeated cordons trigger a GKE auto-repair request.

7.4 Agent failure analysis and relaunch

For job failures caused by code or configuration errors, researchers also need a fast way to diagnose the problem and relaunch the job. After a training crash, diagnostic information is written to a log file and uploaded to the run's file bundles. Researchers can manually dispatch a large language model (LLM) agent to investigate failed jobs and propose fixes. The agent is given the crash log file, run configurations, a checkout of the training code used for the run, and an isolated workspace in our cluster. It has access to several internally defined debug and tracing skills; it can also query our telemetry backends and resubmit the job. Once the investigation is complete, the agent presents the root cause, supporting evidence, and a proposed fix. It then waits for human approval before applying the fix and relaunching the job.

08 Observability and Performance Diagnostics

In GPU infrastructure, failures and performance bottlenecks come from many layers. Our observability also guides autonomous operational decisions like Warden repairs, rather than just powering dashboards. These conditions make correctness, consistency, and granularity important design requirements for our observability system.

8.1 Observability overview

Our observability pipeline comprises multiple metrics and log exporters. Each exporter operates at a different layer and is responsible for a specific subset of the data produced by our infrastructure. Figure 6 provides an overview of this system.

Training metrics span many runs, workers, nodes, and ranks, making accurate metric attribution particularly important. Consistent resource labels and telemetry identifiers connect our application progress with the underlying infrastructure. Our workload dashboard, `gkx-ui`, combines live pod and cluster-membership data with historical metrics and logs to provide a unified view of every active run on our clusters.

The large set of identifiers also creates high-volume, high-cardinality data that traditional log aggregation systems struggle to handle efficiently. We use self-hosted ClickHouse [35] for its high ingestion throughput and fast SQL queries across multidimensional datasets. SQL access also helps with LLM-assisted investigations. [36] Agents can easily create and refine queries to help uncover deep failure and performance insights.

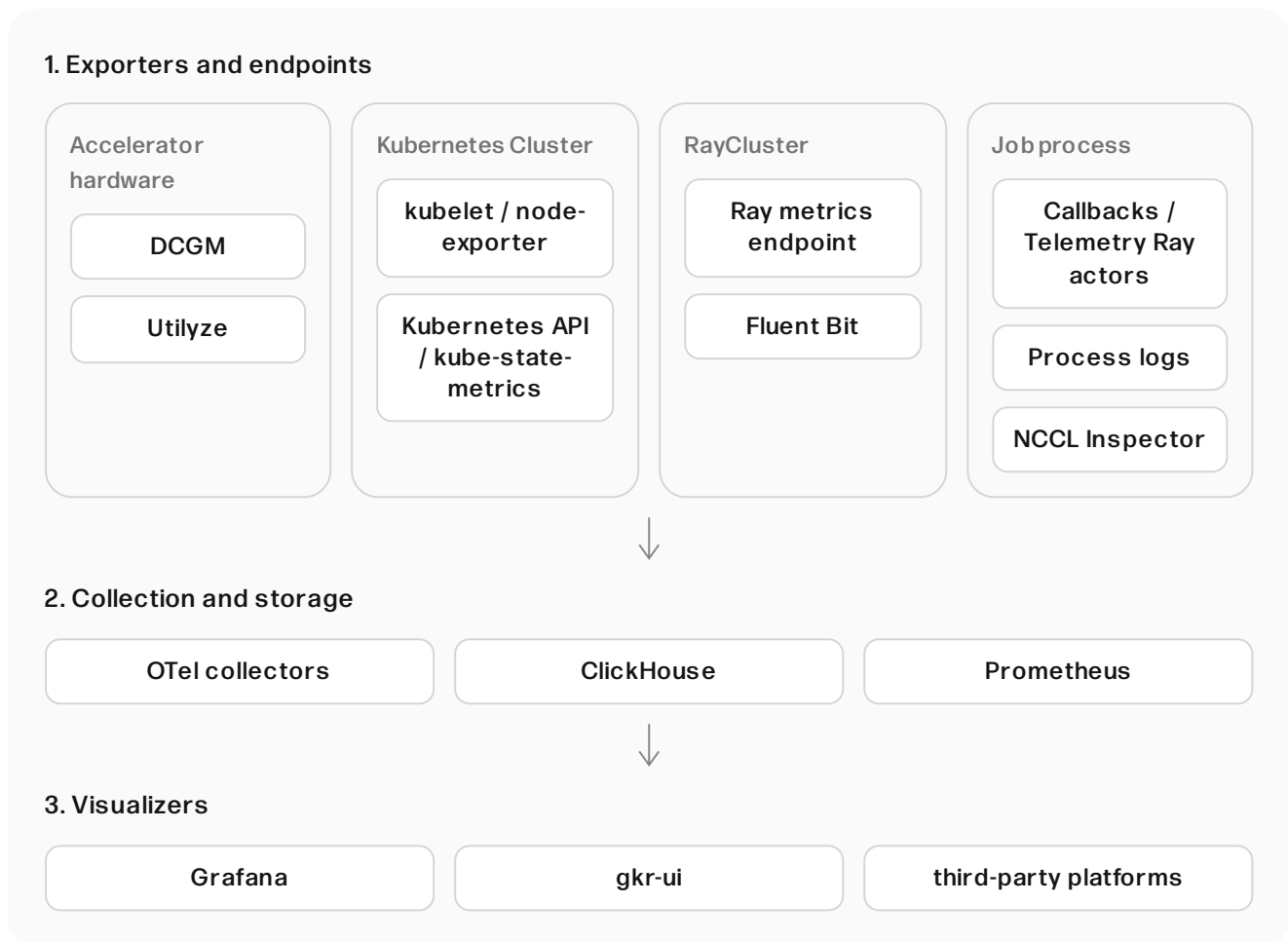


Figure 6: Observability system overview

Collection and visualization components shown above include OpenTelemetry [37], Prometheus [38], and Grafana [39].

8.2 Auxiliary-worker health and performance

FleetMonitor is a Ray actor that aggregates reports from auxiliary services within a run. It measures metrics like queue wait time, processing time, pending work, throughput, and worker health. A training callback sends these summaries to ClickHouse and third-party platforms for aggregation and visualization.

These measurements help distinguish insufficient auxiliary service capacity from slow execution or data transfer. For example, long waiting times may indicate that auxiliary workers need more capacity, while long transfer times may point to communication or serialization overhead. Our engineers use these metrics to guide auxiliary-pool sizing, the ratio of training workers to auxiliary workers, and broader bottleneck investigation.

8.3 NCCL failure and performance diagnostics

The NCCL Inspector [40] records collective durations for each training rank, and Fluent Bit [41] forwards these metrics to ClickHouse. `gkr-ui` then analyzes them to identify persistent patterns of straggling ranks and arrival skew.

In our analysis, we group samples that share the same communication group, collective operation, message size, and algorithm. We model collective duration as waiting time plus transfer time. We also assume that all ranks in the sample group share a similar transfer time. This gives us a simple way to identify the straggling rank: the rank with the shortest measured duration is the last rank to arrive at the collective, and that duration is also an estimate of the transfer time. Other ranks have longer measured durations because they spend time waiting for the slowest rank.

The dashboard applies this method to each rank's collective durations, averaged over approximately 30 seconds. This provides a useful approximation for identifying persistent arrival skew. Let e_i denote the average duration for rank i :

$$\begin{aligned}\text{Average transfer time} &\approx \min_i(e_i), \\ \text{Average waiting time}_i &\approx e_i - \min_j(e_j), \\ \text{Average arrival skew} &\approx \max_i(e_i) - \min_i(e_i).\end{aligned}$$

In addition, the PyTorch flight recorder [42] complements this analysis with a history of collective operations, rank membership, divergence, and failure data. Together, the NCCL Inspector and PyTorch flight recorder provide a coherent picture of NCCL operations in our training system and allow us to diagnose performance issues and failures effectively.

09 Conclusion

A good research platform saves researchers' time and makes efficient use of compute resources. GKR and our Ray training driver give researchers a simple, unified interface for interacting with the infrastructure. Beneath that interface, the Kubernetes platform places workloads based on resource requirements. Inside RayClusters, auxiliary workers handle heterogeneous compute needs, while Ray Train supports the distributed training loop. Cluster reliability is supported by checkpointing, live fault tolerance, and Warden's autonomous maintenance. Finally, unified telemetry connects runs, ranks, workers, pods, and nodes to support in-depth failure and performance analysis. Together, these systems reduce operational overhead and wasted GPU time, allowing a small engineering team to build and operate a platform validated at ten-thousand-GPU scale.

References

- [1] Kubernetes contributors. [Kubernetes](#). *Kubernetes project*, accessed October 6, 2026.
- [2] Philipp Moritz, Robert Nishihara, Stephanie Wang, et al. Ray: A Distributed Framework for Emerging AI Applications. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 561-577, 2018.
- [3] Google Cloud. [Google Cloud Platform](#). *Google Cloud*, accessed October 6, 2026.
- [4] Ray contributors. [KubeRay](#). *Ray documentation*, accessed October 6, 2026.
- [5] Adam Paszke, Sam Gross, Francisco Massa, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.
- [6] Ray contributors. [Ray Train: Scalable Model Training](#). *Ray documentation*, accessed October 6, 2026.
- [7] Argo CD contributors. [Argo CD: Declarative GitOps CD for Kubernetes](#). *Argo CD documentation*, accessed October 6, 2026.
- [8] HashiCorp. [Terraform overview](#). *Terraform documentation*, accessed October 6, 2026.
- [9] Kustomize contributors. [Kustomize: Kubernetes native configuration management](#). *Kustomize project*, accessed October 6, 2026.
- [10] Pallets. [Jinja](#). *Jinja documentation*, accessed October 6, 2026.
- [11] Omry Yadan. Hydra: A framework for elegantly configuring complex applications. *GitHub*, 2019.
- [12] Ari Liberman and Shamel Jacobs. [Understanding Calendar mode for Dynamic Workload Scheduler: Reserve ML GPUs and TPUs](#). *Google Cloud Blog*, 2025.
- [13] Google Cloud. [Google Kubernetes Engine \(GKE\)](#). *Google Cloud*, accessed October 6, 2026.
- [14] Google Cloud. [GPU machines in accelerator-optimized machine family](#). *Compute Engine documentation*, accessed October 6, 2026.
- [15] Nathan Luehr. [Fast Multi-GPU collectives with NCCL](#). *NVIDIA Technical Blog*, 2016.
- [16] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. [Large-scale cluster management at Google with Borg](#). *European Conference on Computer Systems (EuroSys)*, 2015.
- [17] Kueue contributors. [Cluster Queue](#). *Kueue documentation*, accessed October 1, 2026.
- [18] Kyle Aubrey. [Inside the NVIDIA Vera Rubin Platform: Six New Chips, One AI Supercomputer](#). *NVIDIA Technical Blog*, 2026.
- [19] Ray contributors. [Key Concepts: Head Node](#). *Ray documentation*, accessed October 6, 2026.
- [20] Martin Süstrik. [ZeroMQ](#). *The Architecture of Open Source Applications, volume 2*, 2012.
- [21] Ray contributors. [Best practices for deploying large clusters](#). *Ray documentation, Picking nodes*, accessed October 1, 2026.

- [22] Nouamane Tazi, Ferdinand Mom, Haojun Zhao, Phuc Nguyen, Mohamed Mekkouri, Leandro von Werra, and Thomas Wolf. The Ultra-Scale Playbook: Training LLMs on GPU Clusters. *Hugging Face*, 2025.
- [23] Yicheng Lu. Scaling Ray for AI workloads to 10k node clusters. *Anyscale engineering blog*, 2026.
- [24] Shen Li, Yanli Zhao, Rohan Varma, et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proceedings of the VLDB Endowment*, 13(12):3005–3018, 2020.
- [25] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *18th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 230–243, 2001.
- [26] Arthur Douillard, Qixuan Feng, Andrei A. Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc’Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. DiLoCo: Distributed Low-Communication Training of Language Models. *arXiv:2311.08105*, 2023.
- [27] Arthur Douillard, Yanislav Donchev, Keith Rush, et al. Streaming DiLoCo with overlapping communication: Towards a Distributed Free Lunch. *Conference on Language Modeling (COLM)*, 2025.
- [28] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. The Llama 3 Herd of Models. *arXiv:2407.21783*, section 3.3.4 (*Reliability and Operational Challenges*), 2024.
- [29] Ziheng Jiang, Haibin Lin, Yinmin Zhong, et al. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 745–760, 2024.
- [30] Lucas Pasqualin, Less Wright, Iris Zhang, Chien-Chin Huang, Swaminathan Sundararaman, Saransh Gupta, and Raghu Ganti. Reducing Model Checkpointing Times by Over 10x with PyTorch Distributed Asynchronous Checkpointing. *PyTorch blog*, 2024.
- [31] Tristan Rice and Howard Huang. Fault Tolerant Llama: training with 2000 synthetic failures every ~15 seconds and no checkpoints on Crusoe L40S. *PyTorch blog*, 2025.
- [32] Yanli Zhao, Andrew Gu, Rohan Varma, et al. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860, 2023.
- [33] Shawn Wang. Designing a Workflow engine from first principles. *Temporal engineering blog; Systems @ Scale talk walkthrough*, 2021.
- [34] Milind Kukanur. NVIDIA Data Center GPU Manager Simplifies Cluster Administration. *NVIDIA Technical Blog*, 2016.
- [35] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. ClickHouse – Lightning Fast Analytics for Everyone. *Proceedings of the VLDB Endowment*, 17(12):3731–3744, 2024.
- [36] ClickHouse. Why OpenAI chose ClickHouse for petabyte-scale observability. *ClickHouse engineering case study*, 2025.
- [37] OpenTelemetry contributors. Collector. *OpenTelemetry documentation*, accessed October 6, 2026.
- [38] Julius Volz and Björn Rabenstein. Prometheus: Monitoring at SoundCloud. *SoundCloud Backstage blog*, 2015.
- [39] Grafana Labs. Grafana OSS and Enterprise. *Grafana documentation*, accessed October 6, 2026.

- [40] Sirshak Das, Jason Sewall, Giuseppe Congiu, Pavel Shamis, and Gargi Prasad. Enhancing Communication Observability of AI Workloads with NCCL Inspector. *NVIDIA Technical Blog*, 2025.
- [41] Fluent Bit contributors. Fluent Bit. *Fluent Bit documentation*, accessed October 6, 2026.
- [42] Phillip Liu, Uttam Thakore, Junjie Wang, and Justin Yang. Flight Recorder: A New Lens for Understanding NCCL Watchdog Timeouts. *PyTorch blog*, 2026.

BibTeX

```
@misc{reve2026training,  
  title      = {Training Reve 2.0 at scale},  
  author     = {{Reve Research Team}},  
  organization = {Reve},  
  year       = {2026}  
}
```