

AI Factory Training with FlashBlade and Red Hat OpenShift AI

An event-driven, autonomous
LLM fine-tuning reference design,
validated end to end and scalable
from pilot to AI factory

Contents

Executive summary	3
Solution overview	3
Technology overview	4
Key solution features	5
1. Event-driven autonomy	5
2. Quota-aware admission	5
3. Ephemeral distributed training	5
4. Checkpoints that are complete by construction	5
5. Evidence-based promotion	5
6. Storage resilience under load	5
Solution architecture	6
Recommended hardware architecture and design	6
Validated environment	6
Deployment guide	6
Step 1: Prepare the cluster	7
Step 2: Configure FlashBlade storage	7
Step 3: Configure Kueue admission	7
Step 4: Deploy event detection and the trigger	8
Step 5: Deploy the training workload	8
Step 6: Deploy evaluation, gate, and registration	9
Step 7: Secrets and observability	9
Validation results and observations	9
Comparison with traditional approaches	10
Event-driven factory vs. manually launched training	10
FlashBlade and Portworx vs. traditional storage	10
Conclusion	10
Additional resources	10

Executive summary

Enterprises are moving large language model (LLM) fine-tuning from experiments into production, and the operational gaps show up quickly. Four challenges dominate:

- **Utilization:** GPUs stay allocated after training runs finish, and clusters sit idle between jobs while still drawing power and budget.
- **Automation:** Training jobs are launched manually. New data waits on a person, and repeated ad hoc runs risk oversubscribing finite GPU capacity.
- **Durability:** Rank-sharded checkpoints spread across local disks are fragile; an incomplete checkpoint is often discovered only at restore time.
- **Governance:** Models reach production without a defensible record of what data, code, and evaluation evidence produced them.

This reference architecture addresses all four challenges with an event-driven training factory, validated end to end on Red Hat OpenShift AI and Everpure™ FlashBlade®. A new dataset landing in object storage triggers the fine-tuning of a 32-billion-parameter model across a distributed GPU cluster. Kueue admits the job only when its full GPU quota is free. Every rank checkpoints to a shared FlashBlade volume, a held-out evaluation gates promotion, and an immutable lineage record protected by Object Lock ties the production model back to the object that triggered the run. When the job finishes, the cluster tears itself down and the GPUs return to the pool.

The design is a scalable pattern, not a fixed configuration. It runs on the NVIDIA GPUs enterprises deploy for this software stack—NVIDIA GB300/GB200 NVL72 rack-scale systems, HGX B300/B200, RTX PRO, and L40S—with GPU count sized to the model and the job. FlashBlade//S™ carries the design from initial pilot through NVIDIA DGX SuperPOD and AI factory deployments in the 1,000- to 2,000-GPU range, and FlashBlade//EXA™ extends it further.

Solution overview

An AI factory treats training the way manufacturing treats production: inputs arrive, quality gates decide what ships, and the line does not depend on someone standing next to it. The implementation behind this document runs that loop for LLM fine-tuning. It separates four concerns that share one platform: event capture, workload control, distributed computation, and durable evidence.

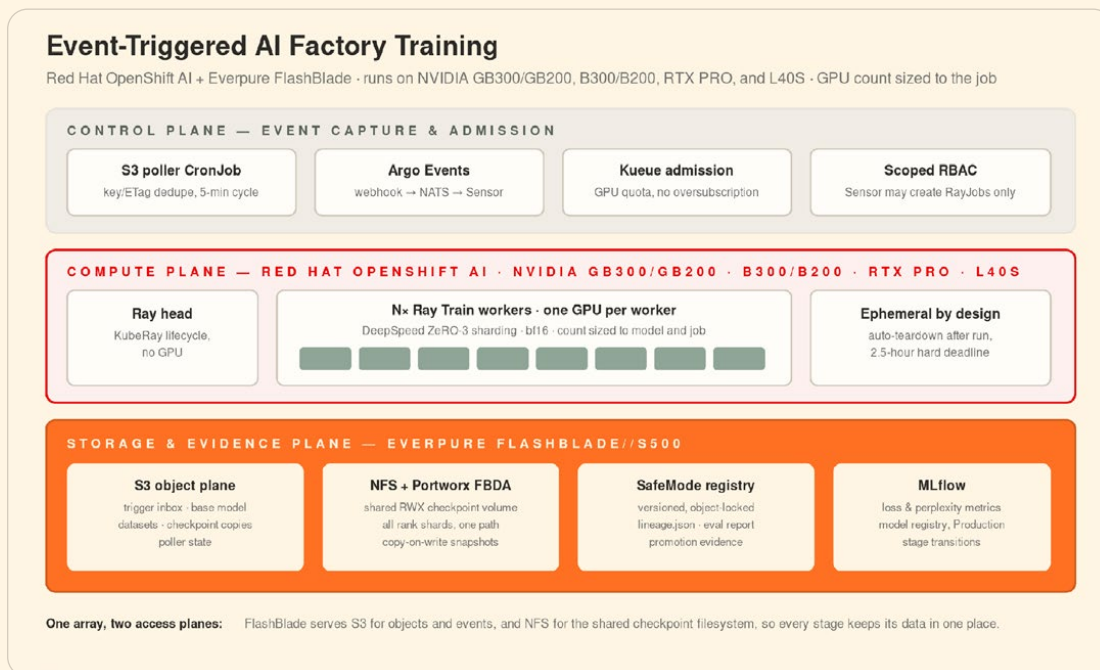


FIGURE 1 The three planes of the architecture. Argo Events and Kueue control when GPUs are consumed, Red Hat OpenShift AI runs the distributed training, and FlashBlade holds the data and evidence.

The storage design is deliberate: FlashBlade serves S3 for the trigger inbox, base models, datasets, checkpoint copies, and registry lineage, and NFS (exposed through Portworx® FlashBlade Direct Access as Kubernetes-native RWX volumes) for the shared checkpoint filesystem. One array carries every stage, and each stage gets the access semantics it needs.

The life cycle begins and ends with zero training GPUs allocated.



Key solution features

1. Event-driven autonomy

A poller CronJob lists the S3 trigger inbox on a five-minute cycle, deduplicates by object key and ETag against the state kept in the same bucket, and posts new objects to an Argo Events webhook. The sensor creates a RayJob with the trigger identity copied into its annotations, so input lineage attaches to the workload before training starts. The trigger inbox is separate from the model and dataset buckets, which keeps the mechanism content-agnostic.

2. Quota-aware admission

Autonomy raises the risk of duplicate or overlapping requests. Kueue converts those into a bounded queue: the workload is admitted only when its full GPU, CPU, and memory request fits the configured quota. A validated lesson is built into this design—an earlier configuration allowed fewer GPUs than the job requested, and the workload queued silently. Size the ClusterQueue to the complete request.

3. Ephemeral distributed training

Ray Train coordinates one DeepSpeed rank per GPU; ZeRO-3 partitions model, gradient, and optimizer state so models far larger than a single GPU's memory train across ordinary, single-GPU workers. Three lifecycle controls make the compute genuinely ephemeral: the RayCluster is deleted the moment the job reaches a terminal state, the RayJob object is retained for 10 minutes for inspection, and a hard deadline (9,000 seconds in the validated run) guarantees a hung job cannot pin accelerators.

4. Checkpoints that are complete by construction

ZeRO-3 writes one shard per rank. Every worker mounts the same FlashBlade Direct Access RWX volume at the same path, the engine saves the full rank directory structure there, and rank 0 uploads the complete tree to S3. The validated run surfaced the failure mode this prevents: a rank-0-only upload silently omitted every shard but its own.

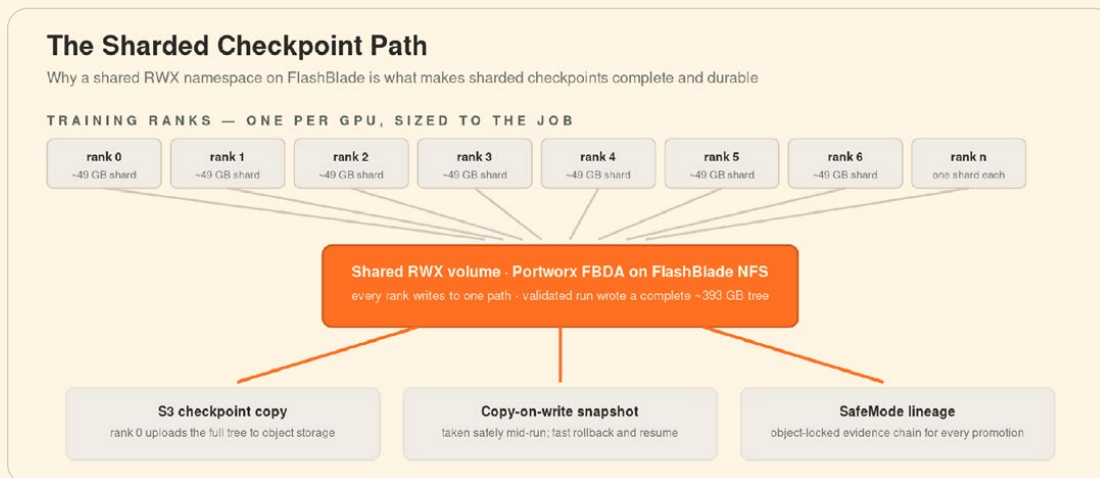


FIGURE 3 The sharded checkpoint path: one shared namespace, and then three durable outputs—S3 copy, copy-on-write snapshot, and SafeMode lineage.

5. Evidence-based promotion

Evaluation, gating, and registration are deliberately separate jobs. A GPU job restores the checkpoint and measures perplexity on held-out samples, and a GPU-free job compares the metric against a configurable threshold and writes a promotion marker only on pass. A registration job then records the lineage—trigger identity, snapshot ID, metrics, code and image identity, and checkpoint path—in the SafeMode-enabled S3 registry and promotes the MLflow model version to production. Both gate branches were exercised against the same measured metric.

6. Storage resilience under load

A copy-on-write snapshot was taken while every rank was actively writing, and it supported the documented restore-and-resume path. The snapshot class uses a retain policy, so deleting the Kubernetes snapshot object does not delete the underlying storage snapshot. The S3 checkpoint copy and the filesystem snapshot serve different purposes: durable off-volume copy versus fast mount and resume.

Solution architecture

Recommended hardware architecture and design

- **Accelerated compute:** NVIDIA GB300/GB200 NVL72 rack-scale systems, HGX B300/B200, RTX PRO, or L40S GPU nodes. One GPU per training worker; worker count is sized to the model and to whether the job is fine-tuning or full training.
- **Storage:** Everpure FlashBlade//S500 serving both S3 and NFS. Size blade count to the workload's model load and retrieval profile; throughput figures are aggregate and scale out with the system.
- **Container data services:** Portworx with FlashBlade Direct Access for RWX volumes and CSI snapshots over FlashBlade NFS.
- **Network:** Standard Ethernet end to end; no proprietary storage clients and no parallel file system to tune.
- **Cluster software:** Red Hat OpenShift on bare metal with the Red Hat OpenShift AI DataScienceCluster providing KubeRay and Kueue.

The same design carries across three deployment scales.

Deployment scale	Storage platform	What it looks like
Pilot and first clusters	FlashBlade//S	The validated design in this document: one array for both storage planes
Enterprise AI factory/SuperPOD (up to ~2,000 GPUs)	FlashBlade//S	Scale-out chassis growth keeps pace with GPU expansion; same namespace and operating model
Frontier-scale AI factory (beyond 2,000 GPUs)	FlashBlade//EXA	1TB/s to 10+ TB/s in a single namespace, with S3 over RDMA

TABLE 2 One reference design across three scales.

Validated environment

The validated environment was a bare-metal Red Hat OpenShift 4.18 cluster with a pilot-scale set of single-GPU workers scheduled across two GPU nodes, FlashBlade//S500 for S3 and NFS, Portworx FlashBlade Direct Access for the shared checkpoint volume, and MLflow for tracking. The training workload fine-tuned Qwen3-32B in BF16 with DeepSpeed ZeRO-3, CPU offload, and pure-PyTorch AdamW.

Deployment guide

This section walks through deploying the pipeline on an existing Red Hat OpenShift cluster. It assumes supported GPUs are installed and the FlashBlade system is reachable from the cluster network.

Deployment repository: The manifests, job specifications, and training code referenced throughout this guide are available to customers and partners in the [GitHub repository](#). Clone the repository, populate the environment values (S3 endpoints and credentials, GPU product labels, quota sizes, and the evaluation threshold), and apply the manifests in the order of the following steps. Steps 2 through 7 can be deployed directly from the repository; Step 1 prerequisites are environment-specific and must be completed first.

High-level flow

- Prepare the Red Hat OpenShift cluster, GPU enablement, and Red Hat OpenShift AI.
- Configure the FlashBlade buckets, FlashBlade Direct Access shared volume, and snapshot classes.
- Configure the Kueue admission quotas.
- Deploy event detection and the RayJob trigger.
- Deploy the training workload.
- Deploy the evaluation, gate, and registration jobs.
- Configure secrets and observability.

Step 1: Prepare the cluster

Install the NVIDIA GPU Operator and Node Feature Discovery Operator so GPU nodes expose `nvidia.com/gpu` resources and product labels. Deploy the Red Hat OpenShift AI DataScienceCluster with the distributed workload components enabled. This provides the supported KubeRay and Kueue operators used in the rest of this deployment guide. Verify that the worker nodes report their GPU product label (used later for placement):

```
oc get nodes -L nvidia.com/gpu.product
oc get datasciencecluster -o yaml | grep -A2 workloads
```

Step 2: Configure FlashBlade storage

Create three S3 buckets with distinct roles, keeping the trigger inbox separate from training data so the trigger mechanism stays content-agnostic.

Bucket/path	Role
<code>s3://ai-val-datasets/incoming/</code>	Trigger inbox—a new object here starts the pipeline
<code>s3://ai-val-datasets/poller-state/seen.tsv</code>	Poller deduplication state (key + ETag pairs)
<code>s3://ft-agentic/</code>	Base models, datasets, and checkpoint copies
<code>s3://ai-model-registry/</code>	Lineage and promotion evidence—enable versioning and SafeMode Object Lock

TABLE 3 Object storage layout.

Expose FlashBlade NFS to Kubernetes through Portworx FlashBlade Direct Access: create a shared RWX PersistentVolumeClaim (fbda-shared in the validated build) that all training workers mount at a common path, and a VolumeSnapshotClass for copy-on-write snapshots. Use a retain deletion policy so storage snapshots survive Kubernetes object deletion:

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: px-csi-fbda-class
driver: pxd.portworx.com
deletionPolicy: Retain
```

Step 3: Configure Kueue admission

Create a ClusterQueue and a LocalQueue in the training namespace, sized to the complete workload request—GPUs, CPU, and memory. The validated failure mode to avoid: a quota smaller than the job request leaves the workload queued with no error. The RayJob carries the queue label:

```
metadata:
  labels:
    kueue.x-k8s.io/queue-name: ft-agentic-lq
```

Step 4: Deploy event detection and the trigger

Because the FlashBlade software level in the validated build returns HTTP 501 for the S3 PutBucketNotification API, detection uses a polling CronJob rather than native bucket notifications. The poller lists the inbox every five minutes, compares keys and ETags against seen.tsv, and posts new objects to the Argo Events webhook. Practical trigger latency is bounded by the polling interval plus scheduling time.

Deploy Argo Events with a three-replica, token-authenticated NATS EventBus, a thin webhook EventSource, and a sensor that creates RayJobs in the training namespace. Scope the sensor's cross-namespace permissions to the minimum:

```
kind: Role
rules:
- apiGroups: ["ray.io"]
  resources: ["rayjobs"]
  verbs: ["create", "get", "list"]
```

The sensor cannot read secrets, create pods, or delete resources (a validated control). Generated RayJobs carry the trigger bucket, key, and ETag as annotations so input identity stays attached to the workload.

Step 5: Deploy the training workload

Deliver the training script through a ConfigMap mounted on the submitter, and submit it as an importable Python module—import the module and call main()—rather than executing the file directly. The direct-script form caused Ray worker pickling and import failures in the validated build. Set the three lifecycle controls on the RayJob:

```
spec:
  shutdownAfterJobFinishes: true
  ttlSecondsAfterFinished: 600
  activeDeadlineSeconds: 9000
```

Inside the training code, four initialization details matter, and all were validated the hard way:

- Initialize DeepSpeed's distributed state explicitly with the NVIDIA Collective Communications Library (NCCL) even though Ray initializes torch.distributed. Without it, DeepSpeed behaves as world_size=1, and each rank tries to hold the full model.
- Instantiate the DeepSpeed config context before loading the model so ZeRO-3 partitions the model during construction instead of materializing all of it on one rank.
- Enable gradient checkpointing with use_reentrant=False; in the validated build, the reentrant path retained gathered parameters during the backward pass and drove per-GPU memory toward out-of-memory territory.
- Use pure-PyTorch AdamW (torch_adam=true) if the JIT-compiled DeepSpeed CPU optimizer fails to build against your NVIDIA CUDA/PyTorch combination.

Checkpointing: Every rank writes into the shared FlashBlade Direct Access path, then rank 0 uploads the complete directory tree to S3. Do not have rank 0 upload only what it can see locally.

Step 6: Deploy evaluation, gate, and registration

Run evaluation as a separate job that restores the checkpoint and performs a forward-only pass over held-out samples (the validated build reused the training script with EVAL_ONLY=1 and RESUME_FROM set to the checkpoint path), writing eval-report.json and logging perplexity to MLflow. The gate is a plain CPU job:

```
if perplexity <= EVAL_THRESHOLD_PPL:
    write promote marker; exit 0
else:
    alert; exit 1 # blocks registration
```

Registration runs only when the promotion marker exists. It writes lineage.json—model name, snapshot ID, global step, perplexity, Git SHA, image digest, dataset source, and checkpoint path—to the SafeMode-enabled bucket and then creates the MLflow model version and transitions it to production. The validated build called create_model_version() directly because a higher-level registry helper was incompatible with the deployed MLflow server version.

Step 7: Secrets and observability

Keep FlashBlade management tokens, S3 credentials, and model-hub tokens encrypted at rest (the validated build uses SOPS with age). Observability is provided through NVIDIA Data Center GPU Manager for live GPU utilization, MLflow for training and evaluation metrics, and the Red Hat OpenShift console for queue and job life cycle. Treat MLflow tracking as best effort—metric logging must not fail the training job when the tracking server is unavailable.

Validation results and observations

The capstone run produced measured results, with both promotion gate branches exercised against the same metric.

Validation area	Observed result
Model	Qwen3-32B fine-tuned in BF16, one DeepSpeed ZeRO-3 rank per GPU
Training signal	Loss fell from 2.03 to 0.67 (~66% reduction)
Checkpoint	Complete sharded checkpoint (~393GB) written to shared FlashBlade Direct Access storage and copied to S3
Held-out evaluation	Perplexity 2.0032 on 64 unseen samples, recorded in eval-report.json and MLflow
Promotion gate	Passed at threshold 2.5; failed and blocked registration at threshold 1.5
Registry	qwen3-fbda-finetune version 1 promoted to production with immutable S3 lineage
Idle state	Zero training GPUs allocated between runs
Snapshot safety	Copy-on-write snapshot taken during active writes; restore-and-resume path validated

TABLE 4 Capstone evidence from the validated run.

Note: The validated implementation is a proof of concept at pilot scale, not a general-purpose orchestration platform. It is not a declarative, multistage directed acyclic graph; evaluation, gating, and registration run as separate jobs. Storage event detection is poll-based at the validated FlashBlade software level.

Comparison with traditional approaches

Event-driven factory vs. manually launched training

Manually launched training leaves two costs on the table: the latency between new data arriving and someone starting a job, and the GPU-hours consumed by clusters that outlive their runs. The event-driven pattern removes both—detection starts the pipeline within the polling interval, admission control queues work against real quotas, and teardown returns accelerators the moment a job ends. The validated steady state is zero training GPUs allocated.

FlashBlade and Portworx vs. traditional storage

Alternative	Where it falls short for this design
Parallel file systems	Comparable feeding rates require proprietary clients, constant tuning, and specialist administration; FlashBlade delivers them on standard Ethernet and NFS
Legacy/scale-out network-attached storage	Cannot sustain massively parallel small reads and saturating checkpoint bursts at the same time
Local NVMe/hyperconverged infrastructure	Rank-sharded checkpoints have no common namespace—the exact silent-incompleteness failure this implementation had to engineer around
Object-only pipelines	No POSIX RWX path for distributed checkpoint collection or fast snapshot-based resume

TABLE 5 Storage alternatives compared.

Conclusion

This reference architecture demonstrates that an autonomous training factory is buildable today from supported shipping components: Red Hat OpenShift AI for orchestration, Kueue for governance, Ray and DeepSpeed for distributed computation, and Everpure FlashBlade with Portworx FlashBlade Direct Access as the data and evidence plane. The validated run proves the full loop—trigger, admission, training, durable checkpointing, gated promotion, and immutable lineage—with GPUs allocated only while work is running. The same pattern, unchanged, scales from a pilot cluster to AI factories in the thousands of GPUs on FlashBlade//S and beyond on FlashBlade//EXA.

Additional resources

- Learn about [Red Hat OpenShift AI](#).
- Learn more about [FlashBlade//S and FlashBlade//EXA](#).
- Learn more about [Portworx FlashBlade Direct Access](#).

Explore Everpure Artificial Intelligence Solutions

Visit Our Website

800.379.PURE

