GPU SMITH
COMPUTE ENGINEERING

GPU SMITH / RESEARCH & PRACTICAL GUIDANCE

Distributed Training Checkpoint Portability: Restore Matrix

> Plan and build private AI compute with GPU Smith. We connect workload requirements with GPU hardware, networking, deployment and operating decisions so your infrastructure fits the work it must perform.

Published by GPU Smith · 2026-11-02 · distributed training checkpoint portability · pytorch distributed checkpoint · checkpoint resharding · fsdp checkpoint world size · deepspeed universal checkpoint · megatron checkpoint conversion · restore correctness · checkpoint manifest

Original article: https://gpusmith.com/articles/en/distributed-training-checkpoint-portability-restore-matrix



In this article

1. Executive Summary
2. Introduction and Background
3. Which State Must Survive
4. PyTorch Distributed Checkpoint (DCP)
5. DeepSpeed ZeRO and Universal Checkpointing
6. Megatron-Core, NeMo and Megatron Bridge
7. Other Stacks and Recovery Layers
8. Feature Comparison
9. The Restore Contract: Manifest, Drills and Classification
10. Performance and Benchmarks
11. Data Analysis and Evidence
12. Implications and Future Directions
13. Frequently Asked Questions (FAQs)
14. Conclusion

Executive Summary

A distributed training job can safely resume on different hardware only when the checkpoint contains more than weights, and only when the restore path has been exercised on the exact topology and software image that will be used. This report separates three claims that save logs routinely blur: a model-only export (weights that can start inference or fine-tuning), a resumable checkpoint (weights plus optimizer moments, scheduler, step, data cursor and random state), and a portable checkpoint (a resumable checkpoint that loads under a different world size, parallelism layout or image). Each claim needs its own evidence. No restore drill was executed for this report; every classification below rests on documentation and source files read on 2026-11-02, and is labeled accordingly.

The documented portability boundary is real but uneven. PyTorch Distributed Checkpoint (DCP) advertises load-time resharding "which enables saving in one cluster topology and loading into another" (docs.pytorch.org), yet the same page warns that "There are no guarantees of Backwards Compatibility across PyTorch versions" (docs.pytorch.org). DeepSpeed's Universal Checkpoint requires a conversion step and a target run configured with the desired parallelism (deepspeed.ai). Megatron-Core's default optimizer format cannot change model parallelism, while its fully reshardable format can (docs.nvidia.com). Frameworks also disagree about what travels with the weights: Megatron Bridge stores iteration count, consumed samples and scheduler state (docs.nvidia.com), while Hugging Face Accelerate does not persist the dataloader position (huggingface.co).

Bitwise reproducibility is a separate and much stricter property. PyTorch states that "Completely reproducible results are not guaranteed across PyTorch releases," (docs.pytorch.org), and cuBLAS says bit-wise reproducibility is not guaranteed across toolkit versions (docs.nvidia.com). A migration approval should therefore require semantic continuity (same completed step, same next batch, same state keys and shapes, optimizer updates and loss inside an owner-defined tolerance) rather than identical bits.

Published measurements are scale-specific: Universal Checkpoint authors report reconfiguration under five minutes for a 1T-parameter model on their storage (arxiv.org), and PyTorch authors report an asynchronous save cutting visible downtime for a 7B model from 148.8 to 6.3 seconds (pytorch.org). Neither transfers to another fleet. The recommended approval gate is an evidence packet containing a manifest, a same-topology baseline, a changed-topology restore, an interrupted-save test, a corrupt-shard test and an incompatible-version test, with pass, unsupported and unproven recorded per row. Loss of in-memory replicas must fall back to long-term storage, as ML Flashpoint's own guidance states (google.github.io).

Introduction and Background

ML platform owners, training engineers and enterprise GPU buyers face a recurring decision: a job has saved checkpoints, and a node pool will be replaced, the GPU count will change or the training image will be upgraded. The question is not whether files were written but whether the next run continues the same training trajectory in the sense the owner cares about. This report is a restore contract for that decision. It is not a storage-capacity sizing guide, a checkpoint-frequency economics model or a fleet failure-rate study, and it offers no universal recovery-time benchmark.

Three definitions anchor the rest of the report.

- **Model-only export:** tensors (and sometimes configuration) sufficient to run or fine-tune the model. Hugging Face's safetensors format, for example, is described as a format "for storing tensors safely" (huggingface.co), and the Trainer option that saves only model weights "prevents resuming training" by dropping optimizer, scheduler and RNG state (huggingface.co).
- **Resumable training state:** model, optimizer, scheduler, loss-scaler, step counter, data cursor and random state, restored together. A checkpoint is resumable only for the software and topology that can interpret every one of those items.
- **Semantic continuity versus bitwise reproducibility:** semantic continuity means the resumed run is the same training process within declared tolerances. Bitwise reproducibility means identical outputs at every step. NVIDIA's deterministic-training guide defines the latter for runs with identical inputs (docs.nvidia.com), and its tests compare two runs of the same configuration, so even the strictest published guarantee does not cover a changed layout.

The publisher of this report, GPU Smith, advises on enterprise GPU infrastructure. It is an adjacent advisor, not a framework vendor, cloud provider or chip manufacturer, and it appears in this report only as the author of the contract described here. Readers with regulated or contractually audited workloads should consult qualified engineering and compliance professionals; no outside reviewer has examined this report.

Evidence labels used throughout: **Documented** means a pinned vendor document or source file states the behavior; **Limit** means a pinned document states a restriction; **Unproven** means no source was found and a drill must establish it; **Pass** is reserved for results from a drill run by the owner. No row in this report is labeled Pass. Versions matter: the distributed checkpoint page examined here is the PyTorch 2.14 page (docs.pytorch.org), while the reproducibility note cited below sits under the 2.15 documentation path (docs.pytorch.org), so every command and claim should be pinned to a version before use.

Which State Must Survive

A resumable checkpoint is an inventory problem. *Table 1* lists the state items a restore contract must account for, who owns each, where the surveyed frameworks document persistence, and how to verify it after restore.

State item	Owner	What the surveyed documentation says	Verification after restore
Model weights	Framework	DCP saves sharded tensors from every rank and loads them in place (docs.pytorch.org)	Key set and shape digest equal to manifest; per-parameter norm within tolerance
Optimizer moments and fp32 master weights	Framework	DeepSpeed keeps fp32 weight and optimizer values in universal form (github.com); Megatron's default optimizer format cannot change model parallelism (docs.nvidia.com)	Optimizer state keys present; first post-restore update matches baseline direction and magnitude
LR scheduler	Application	Megatron Bridge stores scheduler state with iteration and consumed samples (docs.nvidia.com); Accelerate needs scheduler registration (huggingface.co)	Learning rate at step N+1 equals the baseline schedule
Loss scaler	Application or framework	HF Trainer's checkpoint routine saves scaler, RNG and trainer state (raw.githubusercontent.com)	Scale value and growth counter equal manifest
Step and consumed samples	Application	Megatron Bridge persists both; DeepSpeed offers a client state dictionary for extras (www.deepspeed.ai)	Completed-step identifier equals manifest and equals the last finished optimizer step
Data cursor	Application	StatefulDataLoader saves a batch count by default and does not aggregate state across ranks (pytorch.org); Accelerate leaves mid-epoch position to the user (huggingface.co)	First post-restore batch IDs equal the baseline's batch IDs
RNG state	Application	torch RNG state calls are per generator and CPU-only for the default call; Megatron-LM ignores saved RNG when parallelism differs (raw.githubusercontent.com)	Declare RNG as continued, reseeded or not guaranteed; do not claim otherwise
Run configuration and arguments	Application	Megatron-LM asserts fixed model arguments match the checkpoint (raw.githubusercontent.com)	Config hash equal or differences explicitly approved
Format and version metadata	Framework	NeMo records backend and version in metadata.json to pick a loading strategy (docs.nvidia.com); Megatron versioning tracks format, not content (raw.githubusercontent.com)	Manifest records both; mismatch blocks restore
Commit marker	Writer	DCP's finish step writes metadata and marks the checkpoint successful (docs.pytorch.org)	Marker exists, is last-written, and its hash matches manifest

Table 1 shows why "the weights loaded" is not a restore test. Weights are one row of ten, and the rows owned by the application (scheduler, scaler, cursor, RNG) are the ones frameworks most often leave to the training script. DCP loads whatever objects the application places in its state dictionary, and non-tensor data is read

with torch.load and modified in place (docs.pytorch.org). No PyTorch 2.14 page examined states that RNG, sampler or scheduler state is captured automatically; that absence is treated here as Unproven until a drill shows otherwise.

Omitted or unsupported state deserves its own enumeration in any manifest. Candidates seen in the surveyed documentation include dataloader state kept per data-parallel rank, which Megatron Bridge refuses to restore after a data-parallel size change because resuming "would silently change the data order, so it fails loudly instead" (docs.nvidia.com); recipe state stored outside object storage, since a "full recipe checkpoint is not self-contained in object storage" in NeMo AutoModel (docs.nvidia.com); and weights-only initialization, which in Megatron Bridge "does not load optimizer, RNG, dataloader, or scheduler state" (docs.nvidia.com). TorchTitan's initial load defaults to the model only (raw.githubusercontent.com), correct for fine-tuning and wrong for continuing a run.

PyTorch Distributed Checkpoint (DCP)

Capabilities

DCP lets many ranks save and load in parallel. Its reference page says it produces multiple files per checkpoint, at least one per rank, and operates in place, so the model must allocate its storage first. Resharding happens at load: the getting-started recipe states "You can load in the same world size or different world size." (docs.pytorch.org), and explains that DCP needs the target model's sharding information before loading (docs.pytorch.org). On the read side, each rank fetches only the data its local shards require.

The distributed state-dict helpers are the supported route for FSDP, DDP and tensor-parallel modules. The documentation says the helper can process any module parallelized by those PyTorch APIs (docs.pytorch.org), and states the resharding goal as a state dictionary usable "with a different number of trainers and/or different parallelisms". It also explains why the helper exists: optimizer state dictionaries identify parameters by IDs rather than fully qualified names, which can break under some parallelisms. FSDP2 leaves parameter names unchanged after sharding (docs.pytorch.org), which is what makes a key-set comparison across topologies meaningful.

Adoption

DCP is the checkpoint layer other projects build on. Lightning's expert guide says a sharded FSDP checkpoint can be loaded after the world size changed (pytorch-lightning.readthedocs.io); NeMo AutoModel offers a DCP format with "load-time resharding and parallel saving" (docs.nvidia.com); torchtitan documents a single-CPU seed checkpoint that "can be loaded from another job on an arbitrary number of GPUs via DCP resharding" (raw.githubusercontent.com); and Google's ML Flashpoint lists PyTorch DCP as the lowest layer in its stack (google.github.io). A practitioner note by HPC engineer Glenn Klockwood (dated 2026-08-05) describes the idea: each data-parallel replica writes a non-overlapping part of the model weights (www.glennklockwood.com).

Strengths and Limitations

- **Documented compatibility limit:** the save documentation carries the warning that no backward compatibility is guaranteed across PyTorch versions for saved state dictionaries (docs.pytorch.org). Pin the writer's version in the manifest and test the reader's version before migration.
- **Experimental surfaces:** the asynchronous save is flagged "experimental and subject to change" (docs.pytorch.org), the distributed state-dict helpers are flagged experimental, and the collective-free save mode "will change the format of the saved checkpoint and may not be backward compatible".
- **Key agreement:** every rank must supply the same keys; mismatches "may result in hangs or errors" (docs.pytorch.org).
- **Program model:** the recipe states that multi-program multi-data support "is still under development" (docs.pytorch.org). Teams using pipeline layouts where ranks run different programs should treat DCP resharding there as Unproven.
- **Format boundary:** DCP's on-disk format differs from torch.save, so a DCP directory is not a drop-in replacement for a single-file checkpoint.

Commit, durability and integrity

Two events must be kept apart. In the asynchronous API, the returned object carries separate futures for staging and for upload completion (docs.pytorch.org); staging copies state to CPU memory by default before a separate thread saves it, and the visible stall is the device-to-host copy. A step is durable only after the upload future resolves, and the tutorial leaves management of overlapping checkpoints to the user.

For the local-filesystem writer, the 2.14.0 source documents that the target must be an empty or non-existing directory (docs.pytorch.org) and assumes file creation is atomic (docs.pytorch.org). Its finish step writes a temporary metadata file, optionally flushes and fsyncs it, deletes an existing metadata file left from earlier checkpoints, and renames the new one into place (raw.githubusercontent.com). Disabling file sync removes the consistency guarantee after a failure (raw.githubusercontent.com), and the overwrite flag still defaults to true in 2.14.0 (raw.githubusercontent.com). The practical consequence is a rule: write each step to a fresh directory and treat the metadata file as the commit marker, never reuse a directory. On our reading of the 2.14.0 source file, which contains no checksum, CRC or hash handling, shard integrity is not verified at load; the contract therefore adds an external hash manifest. That observation covers one file on one date and must be rechecked for the owner's pinned version.

DeepSpeed ZeRO and Universal Checkpointing

Capabilities

DeepSpeed saves and restores model, optimizer and learning-rate scheduler state through save and load calls, with extra training state carried in a client dictionary (www.deepspeed.ai). Every rank must call the save method, not only rank 0 (raw.githubusercontent.com), and a pointer file named latest records the newest tag (raw.githubusercontent.com). The tag check option verifies that tags agree across ranks (www.deepspeed.ai).

Native ZeRO checkpoints are tied to the data-parallel size with narrow exceptions: in the engine source, automatic repartitioning applies only to fp32 weights of elastic ZeRO stage 1 and 2 checkpoints loaded without optimizer states (raw.githubusercontent.com), and the stage 1 and 2 optimizer repeats that only such weights "can be repartitioned, and only without optimizer states" (raw.githubusercontent.com). Setting the load-optimizer flag to false restores fp32 master weights but starts with fresh optimizer states (raw.githubusercontent.com), a legitimate model-only restart that is not a continuation.

Universal Checkpointing (UCP) is the supported route for changing parallelism. The project's blog describes a distributed representation for saving and a consolidated representation for loading (github.com) and lists pipeline, tensor, data, ZeRO data-parallel, sequence and mixture-of-experts dimensions (github.com). The tutorial's flow is: save a regular ZeRO checkpoint, convert it to universal format, and load it with universal loading enabled (deepspeed.ai). Because UCP does not infer the target layout, the target run still needs the matching DeepSpeed parallelism configuration (deepspeed.ai).

Adoption

UCP has a project blog, a tutorial, a Megatron-DeepSpeed example and an arXiv paper last revised 4 Jul 2025 (arxiv.org). The DeepSpeed configuration reference marks the older elastic checkpoint setting as deprecated and points to Universal Checkpointing for ZeRO-3 (www.deepspeed.ai), and the readthedocs page says universal checkpoints allow changing the number of GPUs (deepspeed.readthedocs.io).

Strengths and Limitations

- **Precision flexibility:** because fp32 weight and optimizer values are kept, a run can move between mixed-precision modes.
- **Stage and pipeline limits in the example:** the Megatron-DeepSpeed README says stage 3 universal checkpoints currently support data parallelism (raw.githubusercontent.com), and checkpoints saved with and without the no-pipeline flag are not mutually convertible (raw.githubusercontent.com).
- **Expert-parallel limits:** the tutorial restricts AutoEP topology changes to data-parallel world-size and expert-parallel size changes (deepspeed.ai), and says UCP does not translate between model families or module paths (deepspeed.ai).
- **Layout-sensitive settings:** ZeRO 1/2 parameter alignment changes the optimizer checkpoint layout, so optimizer checkpoints must be resumed with a compatible padding layout (www.deepspeed.ai).
- **Offline consolidation:** zero_to_fp32 extracts consolidated fp32 weights from ZeRO 1, 2 and 3 checkpoints (raw.githubusercontent.com), a model-only export, not a resume point.

Megatron-Core, NeMo and Megatron Bridge

Capabilities

Megatron-Core's distributed checkpoints are documented as loadable under a different tensor, pipeline or data-parallel configuration from the one that saved them (docs.nvidia.com). NeMo Framework 25.09 states that "contrary to native checkpoints that require the same number of ranks", the distributed format can be read

when job size changes, and that training can resume from checkpoints saved with different tensor and pipeline parallelism degrees (docs.nvidia.com). Megatron Bridge states that tensor, pipeline or data-parallel sizes may change between save and load (docs.nvidia.com).

The optimizer is where the nuance sits. The default data-parallel-reshardable format is fast but "not possible to change model parallelism when using this format"; the fully reshardable format "supports arbitrary changes in model parallelism" (docs.nvidia.com). NeMo 25.11 documents the same split: the default permits resuming with only the data-parallel size changed (docs.nvidia.com), setting the flag lets all parallel dimensions change (docs.nvidia.com), and model weights are always fully reshardable. The consequence for a migration plan is that the flag must have been set at save time, which cannot be fixed retroactively for an already-saved checkpoint.

Adoption

Megatron Bridge's checkpoint stores iteration count, consumed samples and scheduler state. AutoModel documents resuming with a different parallelism strategy (docs.nvidia.com) and keeps optimizer states in DCP format even when the model is written as safetensors (docs.nvidia.com); its consolidated layout loads directly in Hugging Face, vLLM and SGLang (docs.nvidia.com).

Strengths and Limitations

- **Permissive default strictness:** Megatron Bridge's default key-mismatch mode assumes unexpected keys are acceptable and is described as the most permissive (docs.nvidia.com); NeMo exposes a matching load-strictness setting. A restore contract should set strict raising explicitly.
- **Padded layers:** NeMo allows relaxing shape matching for padded layers such as embeddings; any relaxation should be recorded in the manifest.
- **Topology exceptions:** Bridge rejects changing a generalized tensor-parallel topology on load because the SwiGLU layout can reorder rows (docs.nvidia.com), and prescribes a Hugging Face round trip from the original topology (docs.nvidia.com). Optimizer resharding across mixture-of-experts layouts has version-specific support (docs.nvidia.com).
- **Data order:** per-rank dataloader state errors when the data-parallel size differs rather than silently changing the order.
- **Conversion scope:** treat any Megatron-LM to Hugging Face export as a model-only artifact, not a resume point.
- **Version cliff:** optimizer states from older Megatron-Core releases require a re-save path, while model weights from earlier versions load automatically.

Other Stacks and Recovery Layers

Training-loop libraries

Hugging Face Trainer saves model, optimizer, scheduler, scaler, RNG and trainer state in each checkpoint. On resume it normally fast-forwards the data, and the ignore-data-skip option trades speed for a run whose results will not match the interrupted training (huggingface.co). The source logs that reproducibility is not guaranteed

when per-process RNG files are missing because the run was not launched in a distributed fashion (raw.githubusercontent.com). Accelerate expects saved states to come from the same training script (huggingface.co) and leaves mid-epoch resumption to a skip-batches call.

Orbax, in the JAX ecosystem, is explicit about topology: when changing topologies you must specify sharding at restore (orbax.readthedocs.io), and Orbax makes no sharding decisions unless the topology is identical (orbax.readthedocs.io). TorchTitan's v0.2.0 configuration stores the dataloader index for fault-tolerant training so the same batch is not repeated (raw.githubusercontent.com) and lets a run exclude specific keys, such as the dataloader, from loading (raw.githubusercontent.com), which is the documented way to change GPU count at the price of giving up cursor continuity.

Research and recovery layers

ByteCheckpoint (arXiv 2407.20143, version 4 of 2 Apr 2025) uses a parallelism-agnostic representation that reshards at load time (arxiv.org). Its authors note that checkpoints produced by offline resharding scripts are coupled with a specific parallelism (arxiv.org), and that CPU-side state includes RNG, global step and scheduler (arxiv.org). Dataloader token buffers are split or merged when the data-parallel degree changes (arxiv.org), and the authors state that their open-source baselines do not reshard dataloader state (arxiv.org).

ML Flashpoint, maintained by Google, adds an in-memory recovery tier. It "primarily serves for job recovery purposes, not for long-term usage after a training job completes" (google.github.io). Its guidance is placement-sensitive: reuse healthy original machines (google.github.io), because if the job resumes on different nodes "there will be no ML Flashpoint state to recover from" (google.github.io), in which case recovery falls back to long-term storage. A dirty marker written before saving prevents recovery from a partial checkpoint (google.github.io), and replication is always asynchronous (google.github.io). The restore contract inherits this: an in-memory tier raises resilience, but the portability claim must be proven on the durable tier.

Feature Comparison

Table 2 compares what each surveyed stack documents for the migration cases named in the decision. It is a documentation matrix, not a test result. A cell that says Documented means a pinned page states the behavior; every cell still needs a drill in the owner's environment. Cells reading "Expected; drill required" or "Not examined" count as Unproven, and descriptive cells such as "Weights only for TP/PP changes" record a documented Limit.

Stack and layout	Same topology resume	Change data-parallel size or GPU count	Change TP or PP	Optimizer state across a change	Cursor and RNG	Version caveat
PyTorch DCP with FSDP2/DTensor	Documented	Documented	Documented for FSDP, DDP and tensor-parallel modules; MPMD Limit	Documented via state-dict helpers	Application-owned; no automatic capture found	No backward-compatibility guarantee

Stack and layout	Same topology resume	Change data-parallel size or GPU count	Change TP or PP	Optimizer state across a change	Cursor and RNG	Version caveat
DeepSpeed ZeRO native	Documented	fp32 weights only, for elastic ZeRO 1/2	Not documented for native load; use UCP	Optimizer states not repartitioned	Client state	Format changes by release; pin version
DeepSpeed Universal Checkpoint	Documented	Documented	Documented in example; ZeRO-3 example is data-parallel only	Carried in universal form	Not covered by the tutorial text read	Target config must match
Megatron-Core, default optimizer format	Documented	Documented	Limit for optimizer	Weights only for TP/PP changes	Bridge stores step and samples	Format version in metadata
Megatron-Core, fully reshardable optimizer	Expected; drill required	Documented	Documented	Documented; slower than the default format	Per-rank dataloader state errors on data-parallel change	Flag must be set at save time
Orbax (JAX)	Expected; drill required	Must specify sharding	Must specify sharding	Not examined	Application-owned	Pin version
HF Trainer or Accelerate	Documented	Per-process RNG files tied to launch	Not a sharding-aware format	Same script expected	Skip-batches resume	Pin version

Table 2 makes three points. First, resharding of weights is the best-covered case; optimizer, cursor and RNG continuity is where documentation thins. Second, vendors differ in mechanism (resharding at load, conversion before load, or explicit target sharding), so a portability claim must name the mechanism, not only the framework. Third, Megatron's fully reshardable optimizer choice, DeepSpeed's conversion step and DCP's pre-sharded target are decisions that precede the migration, and the contract should record which was used.

The Restore Contract: Manifest, Drills and Classification

Pin the environment and write the manifest

Before any drill, pin the documentation and code versions that define behavior: the PyTorch 2.14 page for DCP, the DeepSpeed release for UCP, the Megatron-Core and NeMo versions for dist-checkpointing. The sources in this report already span the PyTorch 2.14 and 2.15 documentation paths, which is precisely why unpinned commands are unsafe. The manifest is a sidecar file written by the application after the writer's commit marker and verified before any load. A minimal schema (a template, not a tested artifact):

```
{
  "manifest_version": "1",
  "checkpoint_id": "step-0123456",
```

```
  "completed_step": 123456,
  "commit_marker": {"file": ".metadata", "sha256": "..."},
  "software": {"torch": "2.14.0", "cuda": "...", "nccl": "...", "framework": "...", "image_digest": "..."},
  "hardware": {"gpu_arch": "...", "gpu_count": 512},
  "topology": {"world_size": 512, "tp": 8, "pp": 4, "dp": 16, "zero_stage": null, "sharding": "..."},
  "precision": {"param": "bf16", "master": "fp32", "scaler": null},
  "state_keys_sha256": "...",
  "shapes_sha256": "...",
  "files": [{"path": "__0_0.distcp", "bytes": 0, "sha256": "..."}],
  "data_cursor": {"consumed_samples": 0, "per_rank_state": true, "dp_size": 16},
  "rng": {"policy": "continued|reseeded|not_guaranteed"},
  "storage": {"backend": "...", "durable_commit_time_utc": "..."}
}
```

The commit hash and per-file hashes supply the integrity check absent from the examined DCP writer, and the durable commit time records when the asynchronous upload resolved. The storage backend field matters because commit semantics differ: POSIX rename is specified to be atomic (pubs.opengroup.org) and leaves an existing target unaffected on failure (pubs.opengroup.org), but fsync does not necessarily make the directory entry durable (man7.org); S3 updates to a single key are atomic while atomic updates across keys are not possible (docs.aws.amazon.com); only S3 Express One Zone supports atomic rename (docs.aws.amazon.com); and conditional writes can refuse to overwrite an existing key (docs.aws.amazon.com). Google Cloud Storage has strong global consistency (cloud.google.com) but renames objects in flat buckets by copying (cloud.google.com), while hierarchical namespace folders rename atomically (cloud.google.com). For object stores, a manifest object written last with create-if-absent semantics is the commit marker.

Run isolated restore drills

Run each drill in an isolated environment with a copy of the checkpoint, never against the production directory. Recommended order:

1. **Same-topology baseline:** same world size, parallelism, image and storage; this establishes the reference loss trajectory and the run-to-run variance used to set tolerances.
2. **Fewer ranks, then more ranks:** change data-parallel size only, then GPU count with parallelism changes where the framework claims support.
3. **Replacement nodes:** same topology on different hosts; a different GPU model only if the contract claims it, since resharding support does not extend to hardware architecture and cuBLAS results are tied to toolkit version (docs.nvidia.com).
4. **Interrupted save:** terminate during write and during the commit step.
5. **Corrupt shard:** flip bytes in one data file and truncate another.
6. **Incompatible version:** load with a newer or older library and image than the writer.
7. **Loss of in-memory replicas:** discard the memory tier and restore from long-term storage, as ML Flashpoint's fallback prescribes.

Table 3 pairs injected faults with the detection the documentation supports and the classification that applies when detection is not demonstrated.

Injected fault	Method	Expected detection and its basis	If not detected in the drill
Interrupted save before commit	Kill the job before the metadata or manifest is written	Marker absent; AutoModel marks checkpoints resumable only after its marker is cleared (docs.nvidia.com); ML Flashpoint writes a dirty marker	Unsupported for resume from that step
Async staged but not durable	Kill after the staging future, before the upload future	Upload future not resolved	Treat the previous step as latest
Missing shard	Delete one data file	Not documented in the pages read; a read error is plausible but unconfirmed, and mismatched keys across ranks can hang	Unproven; add file-list check from manifest
Corrupt or truncated shard	Flip bytes; truncate	Manifest hash mismatch; storage-side checksums cover transfer only, for example S3 (docs.aws.amazon.com) and GCS CRC32C (cloud.google.com)	Unsupported until hash check is added
Incompatible version	Load with other library or image	DCP offers no cross-version guarantee; Megatron metadata selects loading strategy	Unproven; pin and block
Key or shape mismatch	Rename a parameter; change a padded dimension	Strict raise; Bridge default is permissive	Unsupported until strictness set to raise
Wrong data-parallel size for cursor	Resume with changed data-parallel size	Dataloader state errors where the loader validates its state; otherwise the first batch ID differs	Unproven for cursor continuity
New placement after node loss	Resume on different nodes	No in-memory state; fallback to durable storage	Unsupported without durable copy

Read *Table 3* as a test plan, not a result: whether each cited mechanism triggers in the owner's pinned version and storage is what the drill establishes.

Prove continuity

After each restore, compare against the baseline, in this order:

- **State keys and shapes:** the key digest and shape digest equal the manifest, or each difference is on an approved list.
- **Completed-step identifier:** the restored step equals the manifest's completed step, and the next step is step plus one.
- **Next data batch:** the first N batch identifiers equal the baseline's. Sampler details matter here: DistributedSampler shuffles from a seed and epoch (raw.githubusercontent.com), pads or drops the tail to divide evenly across replicas (raw.githubusercontent.com), and needs set_epoch each epoch or the same ordering is reused (docs.pytorch.org). Changing the replica count therefore changes per-rank membership even when the global permutation is the same.

- **Optimizer updates:** the first updates after restore should match the baseline in direction and magnitude within tolerance; a fresh-optimizer restart will show different early updates, which is why DeepSpeed's flag that restores weights but starts with fresh optimizer states must be recorded.
- **Loss trajectory:** compare a window of steps to the baseline band.
- **Numerical tolerances:** set them with the owner's test policy from measured baseline variance, not from a universal constant. Bitwise equality is not an acceptance criterion across releases or platforms, nor across library configurations. NeMo's guidance on resuming with a different node count is to keep global batch size so steps remain "almost identical", a semantic-continuity statement, not a bitwise one.

Classify and approve

Classify every matrix cell as **Pass** (drill succeeded with evidence attached), **Unsupported** (documentation or drill shows it fails or is excluded) or **Unproven** (no executed test). Unproven cells block migration approval. Require long-term-storage recovery after loss of in-memory replicas, and record the measured recovery time objective (RTO) separately for the durable tier. The evidence packet should contain:

- The manifest for the source checkpoint and for the post-restore checkpoint.
- Pinned versions of documentation, code and image digests.
- The source-to-target matrix with world size, framework, precision and storage backend.
- The fault-injection results (Table 3) with logs.
- The continuity comparison (keys, step, batch IDs, optimizer updates, loss band).
- The restore timing ledger.
- The classification and the named approver.

Approve a migration only when the manifest verifies before load, the commit marker is present and last-written, hashes match, versions are equal or approved, the planned topology change sits inside a Pass row, cursor and RNG policies are declared, the loss band is inside owner tolerance, fallback to durable storage is demonstrated, and no Unproven row remains on the planned path.

Performance and Benchmarks

Published numbers are useful as orders of magnitude and for naming the variables a ledger must capture. None of them substitute for a measurement on the owner's fleet, and each carries its own scope.

- **Asynchronous save downtime:** a PyTorch and IBM Research post reports a 7B model's visible checkpoint time falling from 148.8 to 6.3 seconds, a 23.62x change (pytorch.org), by moving the disk copy off the critical path.
- **Scale:** Meta and Crusoe measured 1856 H200 GPUs with Llama3-70B, HSDP2 and TorchTitan (pytorch.org); GPU blocking was under one second at that scale, and background processing time fell after plan caching and process-based saves (pytorch.org).
- **Local storage tier:** a Meta and Google post measured Llama 3 405B on 432 H200 GPUs (pytorch.org) and describes local checkpointing on node storage.

- **Reconfiguration cost:** the UCP paper reports 1T-model reconfiguration within 3 minutes (arxiv.org), loading from its converted format about 10 seconds slower than a standard load at unchanged parallelism (arxiv.org), and states that total reconfiguration remains under 5 minutes on its storage (arxiv.org).
- **Load-time resharding systems:** ByteCheckpoint reports save and load improvements of up to 9.96x and 8.80x (arxiv.org). Both figures match the FSDP 128-GPU row of the paper's Table 4, which does not name its model. The paper's section 6.1 reports load speedups of 1.41x to 8.80x for loading into unchanged parallelism configurations, so the 8.80x load figure is not a resharding measurement (arxiv.org). The paper also reports an average 54.20x reduction in checkpoint stalls against open-source baselines (arxiv.org).
- **Memory-tier recovery:** ML Flashpoint reports write times up to 120x faster and faster total recovery on four A3-Mega nodes (google.github.io), at the price of tmpfs memory of at least four times the per-node checkpoint size (google.github.io).
- **Remote-storage bandwidth:** the GEMINI paper computes that matching its in-memory result would need 6.4 Tbps of aggregate persistent-storage bandwidth (cdn.amazon.science), against a 62 second iteration time for GPT-2 100B on 16 p4d.24xlarge instances (cdn.amazon.science).

These sources differ in hardware, storage and what is timed, and most measure saving rather than restoring. Their common lesson is that asynchronous or in-memory tiers shorten stalls, not the time to reload from durable storage on a new placement.

Data Analysis and Evidence

A restore contract needs a ledger of measured quantities. The five restore phases to time separately are read, deserialize, reshard, initialize and validate. Two formulas convert the measurements into decision quantities:

- **Scenario replay time:** lost steps multiplied by measured seconds per step, where lost steps are the steps between the last durable commit and the failure.
- **Occupied GPU-hours:** measured GPU count multiplied by restore wall hours, reported separately from productive compute, because a restoring job occupies the allocation without advancing training.

(Hypothetical Example) A 512-GPU job last committed durably at step 123,456 and failed at step 123,576. Measured throughput is 9.5 seconds per step. Lost steps are 120, so scenario replay time is 120 multiplied by 9.5, or 1,140 seconds (0.317 hours). The drill's restore wall time was 0.6 hours. Occupied GPU-hours for the restore are 512 multiplied by 0.6, or 307.2; replaying the lost steps occupies another 512 multiplied by 0.317, or about 162.1. The total of about 469 occupied GPU-hours produced no new training progress. The numbers are invented to show arithmetic and must not be used as benchmarks.

This report contains no restore success rates, failure rates or time-to-recover figures for any fleet. The checkpoint size scales with model size, not cluster size (www.glennklockwood.com), and one storage system described in Meta's Llama 3 paper sustains 2 TB/s with a 7 TB/s peak (arxiv.org); neither converts to the owner's RTO. The 42 minutes that GEMINI computes for a model-state checkpoint to remote storage at 20 Gbps is an analytical figure (cdn.amazon.science).

Implications and Future Directions

Three trends make the portability boundary a current operating decision. First, the recovery stack is layering: DCP at the bottom, then Megatron or NeMo, then memory-tier replication, each documented separately and each with its own caveats. A portability claim at the top inherits the weakest layer. Second, formats are being separated by purpose: sharded training state for resuming, consolidated safetensors for serving. Third, resharding is routine rather than exceptional at scale: the ByteCheckpoint authors report that their production platform identified 1,870 resharding instances during pre-training resumption in six months, alongside further instances from cross-stage reconfiguration and evaluation tasks (arxiv.org).

For procurement, the practical implication is to write restore evidence into acceptance criteria. A buyer replacing a node pool or changing GPU count can ask the platform owner for the evidence packet above before authorizing the move, and can ask a framework vendor which row of the portability matrix a claim corresponds to. For platform playbooks, the contract converts "we saved a checkpoint" into a testable statement with an expiry: it holds for pinned versions, and a new image invalidates it until the drills are rerun. GPU Smith's view, as an advisor, is that restore evidence belongs in procurement acceptance criteria; that is an opinion, not a measured finding, and this report makes no claim about any framework's reliability beyond what is cited.

Open uncertainties remain: application-owned state depends on code the frameworks do not own, optimizer resharding across expert-parallel layouts is version-specific, and no reviewed source offers a universal loss-continuity tolerance.

Frequently Asked Questions (FAQs)

Can a job resume from a checkpoint with a different number of GPUs?

Sometimes, depending on framework and state. PyTorch DCP documents loading in a different world size, Lightning says the same for sharded FSDP checkpoints, and DeepSpeed's native ZeRO path repartitions only fp32 weights of elastic ZeRO 1/2 checkpoints loaded without optimizer states. Treat each as Unproven until a drill passes.

How does PyTorch distributed checkpoint resharding work, and does FSDP load under a different world size?

Each rank reads the least data needed for its local shards, using the target model's sharding information; the planner helper "applies the resharding algorithm and computes the reads needed" (docs.pytorch.org).

What is the difference between a sharded checkpoint and a full state dictionary?

A sharded checkpoint writes per-rank pieces and reshards on load. A full state dictionary consolidates tensors and is easier to share; DeepSpeed's zero_to_fp32 produces consolidated fp32 weights. Neither is resumable unless optimizer, scheduler, cursor and RNG are included.

How is a DeepSpeed or Megatron checkpoint converted to a different parallelism?

For DeepSpeed, save a regular ZeRO checkpoint, convert it to universal format, then load it with a target configuration that sets the desired parallelism. For Megatron, use the distributed format and enable the fully reshardable optimizer setting at save time; for topologies the format rejects, round-trip through Hugging Face weights with Megatron Bridge. Because that route moves weights, treat optimizer state as lost (our inference, to be confirmed in a drill).

How is restore correctness verified, and what does it cost?

Compare keys and shapes, completed step, next batch identifiers, optimizer updates and a loss window to a same-topology baseline, with tolerances set by the owner; bitwise equality is not guaranteed across releases or platforms. The UCP paper reports about 10 seconds of extra load overhead at unchanged parallelism; measure your own.

Conclusion

Checkpoint portability is a property of a specific save, a specific restore path, and a specific pair of environments, not of a framework name. Weights are the easiest part to move; optimizer state, scheduler, scaler, data cursor and random state determine whether the next step continues the same training process, and their persistence is documented unevenly across the surveyed stacks. Documentation supports load-time resharding in PyTorch DCP, conversion-based reconfiguration in DeepSpeed, and format-dependent optimizer resharding in Megatron-Core and NeMo, alongside explicit warnings about version compatibility, permissive default strictness and placement-sensitive memory tiers.

A fleet migration should therefore be authorized on restore evidence: a manifest that pins versions and records a commit marker and integrity hashes, drills for each topology and fault the plan depends on, a continuity comparison against a baseline, and a Pass, Unsupported or Unproven label on every cell. Until a cell is Pass for the planned path, its status is Unproven and the fallback is the last durable checkpoint on long-term storage.

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. https://docs.pytorch.org/docs/2.14/distributed.checkpoint.html
2. https://deepspeed.ai/tutorials/universal-checkpointing
3. https://docs.nvidia.com/megatron-core/developer-guide/0.18.1/api-guide/core/dist_checkpointing.html
4. https://docs.nvidia.com/nemo/megatron-bridge/latest/training/checkpointing.html
5. https://huggingface.co/docs/accelerate/usage_guides/checkpoint
6. https://docs.pytorch.org/docs/2.15/notes/randomness.html
7. https://docs.nvidia.com/cuda/cublas/index.html
8. https://arxiv.org/pdf/2406.18820
9. https://pytorch.org/blog/reducing-checkpointing-times

10. https://google.github.io/ml-flashpoint
11. https://huggingface.co/docs/safetensors/index
12. https://huggingface.co/docs/transformers/main_classes/trainer
13. https://docs.nvidia.com/megatron-core/developer-guide/latest/user-guide/deterministic-training.html
14. https://github.com/deepspeedai/DeepSpeed/blob/master/blogs/deepspeed-ucp/README.md
15. https://raw.githubusercontent.com/huggingface/transformers/main/src/transformers/trainer.py
16. https://www.deepspeed.ai/getting-started
17. https://pytorch.org/data/beta/torchdata.stateful_dataloader.html
18. https://raw.githubusercontent.com/NVIDIA/Megatron-LM/core_v0.15.0/megatron/training/checkpointing.py
19. https://docs.nvidia.com/nemo-framework/user-guide/25.09/nemotoolkit/checkpoints/dist_ckpt.html
20. https://raw.githubusercontent.com/NVIDIA/Megatron-LM/core_v0.15.0/megatron/core/dist_checkpointing/core.py
21. https://docs.nvidia.com/nemo/automodel/latest/guides/checkpointing.html
22. https://raw.githubusercontent.com/pytorch/torchtitan/v0.2.0/torchtitan/config/job_config.py
23. https://docs.pytorch.org/tutorials/recipes/distributed_checkpoint_recipe.html
24. https://docs.pytorch.org/docs/2.14/distributed.fsdp.fully_shard.html
25. https://pytorch-lightning.readthedocs.io/en/stable/common/checkpointing_expert.html
26. https://raw.githubusercontent.com/pytorch/torchtitan/v0.2.0/docs/checkpoint.md
27. https://www.glennklockwood.com/garden/checkpointing
28. https://raw.githubusercontent.com/pytorch/pytorch/v2.14.0/torch/distributed/checkpoint/filesystem.py
29. https://raw.githubusercontent.com/deepspeedai/DeepSpeed/master/deepspeed/runtime/engine.py
30. https://www.deepspeed.ai/docs/config-json
31. https://raw.githubusercontent.com/deepspeedai/DeepSpeed/master/deepspeed/runtime/zero/stage_1_and_2.py
32. https://arxiv.org/abs/2406.18820
33. https://deepspeed.readthedocs.io/en/latest/model-checkpointing.html
34. https://raw.githubusercontent.com/deepspeedai/Megatron-DeepSpeed/main/examples_deepspeed/universal_checkpointing/README.md
35. https://raw.githubusercontent.com/deepspeedai/DeepSpeed/master/deepspeed/utils/zero_to_fp32.py
36. https://docs.nvidia.com/nemo-framework/user-guide/25.11/nemotoolkit/checkpoints/dist_ckpt.html
37. https://docs.nvidia.com/megatron-core/developer-guide/0.15.0/api-guide/moe.html
38. https://orbax.readthedocs.io/en/latest/guides/checkpoint/checkpointing_pytrees.html
39. https://arxiv.org/abs/2407.20143
40. https://arxiv.org/html/2407.20143v4
41. https://google.github.io/ml-flashpoint/overview
42. https://pubs.opengroup.org/onlinepubs/9799919799/functions/rename.html
43. https://man7.org/linux/man-pages/man2/fsync.2.html
44. https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html
45. https://docs.aws.amazon.com/AmazonS3/latest/userguide/directory-buckets-objects-rename.html

46. https://docs.aws.amazon.com/AmazonS3/latest/userguide/conditional-writes.html
47. https://cloud.google.com/storage/docs/consistency
48. https://cloud.google.com/storage/docs/hns-overview
49. https://docs.aws.amazon.com/AmazonS3/latest/userguide/checking-object-integrity.html
50. https://cloud.google.com/storage/docs/data-validation
51. https://raw.githubusercontent.com/pytorch/pytorch/v2.15.0/torch/utils/data/distributed.py
52. https://docs.pytorch.org/docs/stable/data.html
53. https://pytorch.org/blog/6x-faster-async-checkpointing
54. https://pytorch.org/blog/distributed-checkpoint-efficient-checkpointing-in-large-scale-jobs
55. https://arxiv.org/pdf/2407.20143
56. https://cdn.amazon.science/29/31/6523473f48e4af52252bac56ef51/gemini-fast-failure-recovery-in-distributed-training-with-in-memory-checkpoints.pdf
57. https://arxiv.org/pdf/2407.21783

THE PUBLISHER

About GPU Smith

GPU Smith provides independent engineering and research for private AI infrastructure. We help technical buyers and operators reason about GPU workload sizing, hardware selection, procurement, deployment and inference operations, from the compute system to the networking, power and cooling requirements around it.

Start with the workload

Useful infrastructure decisions begin with the models, data, throughput, latency and operating constraints a team actually has. GPU Smith helps connect those requirements with system design and deployment choices. The goal is a privately controlled AI environment whose capacity and operational demands are understood before a purchasing decision.

Hardware and supplier research

Our public reference library covers GPUs, complete systems and networking components. The server-vendor directory and manufacturing research help teams investigate suppliers, compare options and follow sources behind technical claims. We distinguish manufacturer specifications from measured benchmarks and advertised capabilities from tested configurations. Reference pages are research resources, not a live inventory listing or a binding equipment quote.

Deployment and operations

GPU Smith's engineering scope includes infrastructure integration, networking, power, cooling and the practical operation of inference workloads. Our published articles explain the tradeoffs behind deployment, procurement and ongoing operations so teams can ask better questions and document decisions.

Work with GPU Smith

Explore the hardware library, GPU references, networking references and vendor research. Contact GPU Smith with your workloads and deployment constraints to discuss a project and confirm current scope, pricing and availability.

Inclusion of a manufacturer or product in our research does not imply a partnership, certification or endorsement.

Website: https://gpusmith.com

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.