

NeMo-DCR: Bit-Exact Delta-Compressed Refit for Scalable Agentic RL at Trillion-Parameter Scale

Songlin Jiang^{1,2*}, Zhiyu Li², Terry Kong², Yu Yao², Youngeun Kwon²,
Bernard Nguyen², Ashwath Aithal² and Mario Di Francesco¹

¹Aalto University, ²NVIDIA

Abstract. Agentic reinforcement learning (RL) disaggregates training from rollout, so each policy update must reach the rollout clusters before the next batch. Transferring a full 1T checkpoint for such weight synchronization (*refit*) takes 87.5 min between two AWS regions. Measurements of BF16 training show that about 1% of weights change their stored values per step. Recent systems exploit this sparsity but fall short on placement, exactness, or efficiency: they reimplement placement rules, assemble full tensors, rebuild values arithmetically, or use a cross-cluster collective, and none fully recovers from mid-refit failures.

We present *NeMo-DCR* (Delta-Compressed Refit), which sends only changes yet is bit-exact: receivers obtain the same parameter and buffer bits as a dense refit. For placement, fixed affine mappings project changes from training shards into the checkpoint’s *canonical coordinates*, residual conversion covers the other changes, and the serving runtime’s native loader places all changes in receiver storage. For exactness, compressible XOR masks carry affine changes whose projection and loader preserve stored bits, and overwrites carry the others. Receivers apply both in place, retries overwrite partial writes, and a joint commit binds the policy to the baseline for the next delta. For efficiency, object storage or a relay tree streams payloads during delta construction, without a cross-cluster collective. Even at 3% and 5% change rates, NeMo-DCR refits of 30B–1T models are 12–40× faster than a transport-only full-checkpoint reference. A 1T relay-tree refit at 3% takes 150 s instead of 87.5 min, making refits practical for cross-cluster agentic RL at trillion-parameter scale.

■ Documentation: [NeMo RL weight refit guide](#) 🔗 Code: [NeMo RL pull request #2444](#)

1. Introduction

Agentic reinforcement learning (RL) spends over 70% of its wall-clock time in rollout, so rollout is disaggregated from training and runs on serving clusters with suitable hardware and parallel layouts [6, 7]. Before the next rollout batch, a *refit* must deliver the updated policy to the *receivers*, the rollout ranks that generate responses with a serving runtime [15]. Disaggregation also lets rollout run on idle serving GPUs [6] or in other datacenters [40, 41], where every refit must cross a wide-area network.

A cross-cluster transfer of a 120B model’s 247.2 GB checkpoint through object storage takes 750 s, and at the 1T scale of models like Kimi K2 [14], such a transfer takes 87.5 min (Figure 1). Yet in the six models of Figure 2, only 0.6–1.2% of training-side source elements change their bfloat16 (BF16) values per step. We call this fraction the *element change rate*. With only 8 significant bits, BF16 rounds most optimizer updates away. Other reports give element change rates of 1.5–2.0% in Qwen3 RL [26] and average deltas of 1.98% of a 1,024 GiB checkpoint [5]. A full-checkpoint transfer thus spends 98–99% of its bytes on unchanged elements.

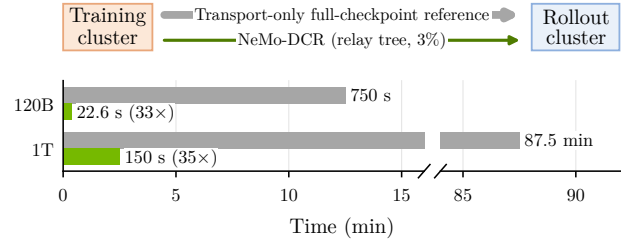


Figure 1 | NeMo-DCR relay-tree refit at a 3% element change rate versus the transport-only full-checkpoint reference: 22.6 s instead of 750 s at 120B and 150 s instead of 87.5 min at 1T (§§ 8.5 and 8.6).

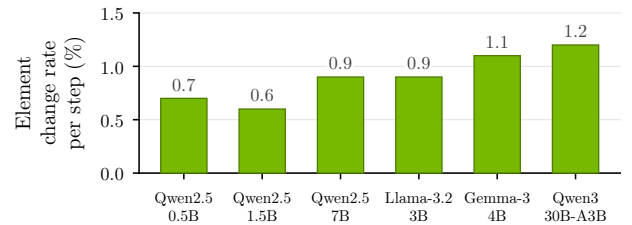


Figure 2 | Element change rate per optimizer step, averaged over each model’s first five steps with the setup of Appendix E: at most 1.2% in every model.

*Work conducted during an internship at NVIDIA.

A refit that sends only the changed 1% faces three challenges: placement, exactness, and efficiency.

First, placement: sparse source changes do not tell receivers where or what to write. A training-shard location does not map directly to a receiver’s storage location: refitting may gather, split, or permute tensors between training and rollout layouts [26]. A change to a shared scale factor can also alter converted values even when other source values stay unchanged [25]. Some systems reimplement the serving runtime’s placement rules [6, 26], and others extract a delta after full-tensor assembly and conversion, which handles both problems but pays for both steps [11].

Second, exactness: a smaller transfer must still deliver the exact updated bits, because rollout–training mismatch is known to destabilize RL [28]. Arithmetic reconstruction can introduce floating-point rounding errors [26], while absolute overwrites avoid rounding [19] but resend unchanged bits within changed values. Failures also threaten exactness: applying updates in place avoids a separate receiver-side baseline copy, but an interrupted refit can leave a mixture of old and updated values that matches neither policy version. Because a delta applies only to the version it was computed from [19], recovery must complete the interrupted refit before rollout resumes and keep receivers aligned with the source baseline that the next refit compares against.

Third, efficiency: delivery must overlap the other refit stages without a cross-cluster collective. The next rollout batch waits for the complete update, so without overlap, delta construction and application can offset the transfer savings. A cross-cluster collective couples receiver membership to the training cluster and introduces timeouts upon failure [23]. Some deployments share weights only through object storage [3], and others have direct links between clusters [40], so delivery must support both.

Existing systems meet these challenges only in part: they either send every weight or lack exactness, native placement, decoupled delivery, or recovery, so none meets all five requirements in Table 1. We present NeMo-DCR, a system for *bit-exact delta refit* that meets all five: it sends only changes, yet every receiver obtains the same parameter and buffer bits as a dense refit that sends every weight. The key idea is to describe changes in *canonical coordinates*, the tensor names and indices of the Hugging Face checkpoint format that both training and serving already support. Fixed affine index mappings project changes directly into these coordinates and cover over 96% of the weight bytes in the mixture-of-experts (MoE) checkpoints. The serving runtime’s native loader then maps these coordinates to receiver storage, much as a

page table maps virtual to physical addresses. Building on canonical coordinates, this paper makes four contributions that address these three challenges:

1. **Direct projection and residual conversion.** One training rank per shard acts as its *owner*, which detects changes in stored values and projects affine changes into canonical coordinates. Residual conversion compares converted tensors with a distributed residual baseline to cover the other changes, including those from shared scale factors. Avoiding full-tensor assembly and conversion for affine changes makes Qwen3 delta construction 1.08–1.16× faster at 3% and 5% element change rates (§§ 4 and 8.3).
2. **Mixed XOR/overwrite encoding with compression.** XOR encodes non-overlapping affine changes whose projection and loader preserve stored bits, while overwrites encode all others. Both are exact, and mixed encoding cuts Qwen3 payload bytes by 38–40% versus overwrites (§§ 5, 8.2, and 8.3).
3. **Recoverable in-place refits.** Receivers leave placement to the native loader and apply updates in place, keeping no separate receiver-side baseline copy. Retries overwrite the complete changed set, repairing partial writes. After all current receivers apply the updates, a joint commit binds the policy to its source baseline, which owners then update in place. Proposition 1 guarantees bit-exactness. With receivers killed mid-refit, both NeMo-DCR transports follow the mean reward and KL trajectories of dense NCCL refits (§§ 6, 8.1, and 8.4).
4. **Pipelined delivery without a cross-cluster collective.** Payloads travel over object storage or a relay tree as the delta is built and, in synchronous RL, applied. In asynchronous RL, rollout continues serving requests until the delta has been transferred and staged, then pauses while receivers apply it. At 3% and 5%, relay-tree transport overlaps delta construction, and the transport lower bound accounts for 77–94% of the refit latency (§§ 7 and 8.7).

Together, these contributions make NeMo-DCR refits of 30B–1T models 12–40× faster than the transport-only full-checkpoint reference (§§ 8.5 and 8.6), even in stress cases whose 3% and 5% element change rates exceed every rate in Figure 2. The same code handles Qwen3 and hybrid Mamba-Transformer Nemotron models without model-specific logic. At 1T and 3%, the relay tree takes 150 s instead of 87.5 min (Figure 1), making refits practical for cross-cluster agentic RL at trillion-parameter scale.

Table 1 | Refit approaches scored against the five requirements of § 2.3. Only NeMo-DCR meets all five.

Approach	Systems	Bit-exact G1	Delta G2	Native G3	Decoupled G4	Recovery G5
Ⓐ Dense collective refit	NCCL-Reshard [23] checkpoint-engine [20]	✓ ✓	– –	(✓) ✓	– –	(✓) (✓)
Ⓑ Full-checkpoint transfer	TensorHub [40]	✓	–	(✓)	✓	(✓)
Ⓒ Arithmetic reconstruction	ROSE [6]	–	✓	–	✓	–
	AReaL-DTE [26]	–	✓	(✓)	✓	(✓)
	AuroraRL [30]	–	✓	–	✓	(✓)
Ⓓ Absolute overwrites	PULSE [19]	(✓)	✓	–	✓	(✓)
	SparseRL-Sync [11]	(✓)	✓	✓	–	–
	verl [36]	(✓)	✓	✓	–	–
Bit-exact delta refit	NeMo-DCR	✓	✓	✓	✓	✓

✓ means met, (✓) partly met, and – not met or not described in the cited source. Details: Appendix F.

2. Motivation and Requirements

2.1. Refit across layouts and versions

Training systems and serving runtimes store the same policy in different layouts. Training systems shard parameters, gradients, and optimizer state to fit into GPU memory while maximizing update throughput [21, 29]. Serving runtimes organize weights for inference and maintain request state such as a paged key-value (KV) cache [15]. RL systems that coordinate the training and rollout sides must bridge the two layouts [10, 13, 32]. The two sides choose their parallel degrees independently, so one training shard can overlap several rank-local slices on receivers, and one slice can combine parts of several shards (Figure 3).

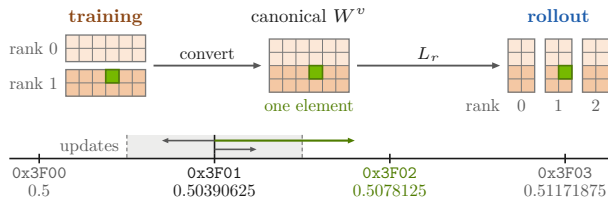


Figure 3 | Top: one tensor in the training, canonical, and rollout layouts. Bottom: BF16 values near 0.5, labeled by their stored bits in hexadecimal. Values in the shaded interval round to 0x3F01, and only an update crossing its boundary changes the stored bits.

Both layouts map to canonical tensors: the model’s tensors in the Hugging Face checkpoint’s naming and layout, independent of either cluster’s sharding. Let W^v denote the canonical tensors at version v , including versioned buffers. A receiver r does not store W^v directly. Its native loader L_r is the serving runtime’s own weight loader and may split, combine, skip, transpose, or cast tensors before writing rank-local storage $P_r^v = L_r(W^v)$. For example, vLLM fuses the query, key, and value projections and splits routed experts across ranks. Each sequence of such operations is a

loader path, and we consider loader paths whose output elements each come from one canonical element.

A refit must bring each receiver to $P_r^{v+1} = L_r(W^{v+1})$, the result of a *dense refit*. Dense collective refit and full-checkpoint transfer produce this result by sending every weight, and Table 1 lists them as approaches A and B. We call the update to version $v+1$ a *transition*, which may take several attempts. Each attempt updates the current receivers, called its *required receivers*.

2.2. Few BF16 values change per step

How many values a refit must update depends on training: an update changes a stored BF16 value only when it crosses a rounding boundary (Figure 3), so larger learning rates and early training steps change more stored values. Yet even in the first five steps of Group Relative Policy Optimization (GRPO) [31], the six models in Figure 2 show only 0.6–1.2% average element change rates, counting each unique source element once. Appendix E details the setup, which follows an agentic software-engineering (SWE) RL recipe with a constant 10^{-6} learning rate.

2.3. Requirements and related work

To exploit this sparsity, a refit must meet five requirements that make placement (G3), exactness (G1 and G5), and efficiency (G2 and G4) concrete (Table 1). **G1 Bit-exact:** starting from receiver storage P_r^v , a delta refit must produce the same parameter and buffer bits as a dense refit. **G2 Delta:** transfer volume must scale with the emitted changed values, their locations, and payload metadata. **G3 Native:** the native loader must determine placement, with no model-specific placement rules in the refit system. **G4 Decoupled:** no collective may span the training and rollout clusters. **G5 Recovery:** receivers must

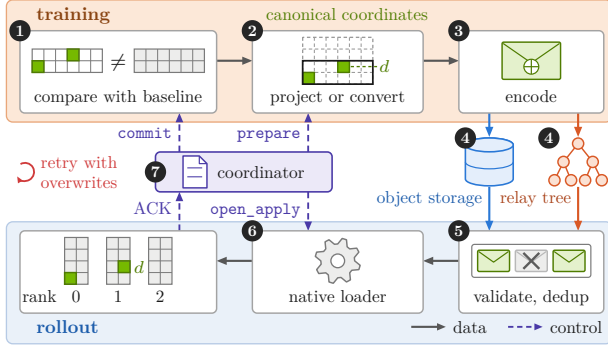


Figure 4 | NeMo-DCR architecture and the numbered steps of a refit. The coordinator calls **prepare** to start an attempt and **open_apply** to let receivers apply payloads and then acknowledge (ACK).

suppress duplicate payloads, partial writes must be recoverable, and one authoritative commit record must bind each new policy version to its source baseline after every required receiver holds its new storage P_r^{v+1} . No existing system in Table 1 fully meets G5, and each fully meets at most two of the five requirements.

Other systems that ship policy weights to serving runtimes are limited in scalability or efficiency: they send full checkpoints, rebuild full weights from deltas, or support only unsharded runtimes. INTELLECT-2’s SHARDCAST relays checksummed full checkpoints in pipelined shards [27]. In Composer 2 and Fireworks, trainers upload per-step deltas to shared storage, from which inference clusters rebuild full weights [3, 5]. TRL’s delta sync likewise rebuilds full tensors from a CPU snapshot [12], and the slime framework applies XOR or overwrite deltas to a full local checkpoint and reloads it at every refit [33]. vLLM patches runtime parameters in place, but doesn’t support any parallelism [37].

Related techniques target training traffic, fine-tuning, file transfers, or storage rather than policy refits: gradient compression [17, 38], low-rank adaptation [9], delta file transfer and lossless weight compression [8, 34], and compression of fine-tuning deltas and checkpoints [4, 16, 18, 39].

3. Design Overview

Canonical coordinates split a refit into delta construction and receiver-side placement. Figure 4 numbers the seven steps: owners construct deltas, a transport delivers them, receivers apply them, and the *coordinator*, which drives each attempt, publishes the policy with a joint commit. The next four sections (§§ 4–7) explain these steps in detail and develop the four contributions in turn: projection and conversion, encoding, application and recovery, and delivery.

Projection and conversion (1–2, §4). Shard owners compare their weights with the source baseline and directly project changes covered by affine mappings, which are fixed and convert each source element independently. The rest are residual: their canonical tensors must be constructed first (contribution 1).

Encoding (3, §5). The codec encodes a change as an XOR entry when projection and the loader path preserve stored bits and no other write overlaps it, and as an overwrite entry otherwise (contribution 2). The first three steps together meet G1 and G2.

Application and recovery (5–7, §6). Each receiver validates and deduplicates payloads, then applies their XOR or overwrite entries in place through the native loader. Failed attempts are retried with overwrites. After every required receiver acknowledges, the coordinator’s joint commit binds the policy version to its source baseline (contribution 3). These steps together meet G1, G3, and G5.

Delivery (4, §7). Object storage or the relay tree delivers the entries in payloads without a cross-cluster collective (G4). Streaming overlaps delivery with delta construction and, in synchronous RL, also with application (contribution 4).

Example. Suppose that a BF16 element changes its stored bits from 0x3F01 to 0x3F02 in a matrix sharded by rows in training and by columns in rollout. Its owner detects the change (1), projects it to its canonical destination d (2), and emits the XOR mask 0x0003 in a compressed payload (3). The payload reaches every rollout node (4), and each node’s receivers validate it (5). Each receiver whose column slice holds d intercepts the loader’s final storage copy in order to XOR 0x0003 into 0x3F01 (6). The joint commit publishes the updated policy, and the owner’s baseline becomes 0x3F02 (7).

Implementation. NeMo-DCR is about 7K lines of Python. It builds on Megatron Bridge conversion [25] and the vLLM native loader [15].

4. Direct Projection and Residual Conversion

In 1–2 of Figure 4, NeMo-DCR compares each training shard with its source baseline in place, on the shard owners: direct projection avoids full-tensor assembly and conversion for affine changes, and residual conversion covers the other changes.

The changes that an owner can directly project depend on the *conversion tasks* through which the Megatron Bridge library converts training shards to canonical tensors [25]. *Affine* tasks convert

each source element independently and deterministically with the fixed affine index mappings of supported mapping classes, such as those for replicated and row- or column-split shards. Tasks for fused query/key/value (Q/K/V) source tensors and other tasks outside the supported mapping classes are *residual* and require constructing canonical tensors. Affine mappings cover 96.3% of the weight bytes in the Qwen3-30B-A3B checkpoint and 97.0% in the Nemotron-3-Ultra-550B-A55B checkpoint, both MoE models, as Appendix D details.

4.1. Ownership and change detection

In ❶, only one owner compares each unique source element against the source baseline, because data-parallel training keeps its replicas bitwise identical. A stable name hash balances ownership across data-parallel ranks (Figure A.2) and keeps it fixed across the attempts of a transition.

Owners detect changes against the committed source baseline B^v . It contains a *task tracker* for each conversion task, holding the version- v values of the task’s source shard, and a distributed *residual baseline* of canonical tensors for residual tasks. Because each unique source element has one owner, the trackers together hold one copy of the source weights. For task T , let S_T^{v+1} be the candidate shard, A_T^v its tracker, and J_T its owned indices. Let $I(X)_j$ denote the stored bit pattern of X at j . The changed indices are

$$C_T = \{j \in J_T \mid I(S_T^{v+1})_j \neq I(A_T^v)_j\}.$$

Comparing stored bits detects all changes, even those that floating-point equality would miss (Figure A.2).

Baseline memory. The baseline uses host memory, not GPU memory. Beyond one copy of the source weights in the trackers, it holds the canonical tensors of residual tasks, which are 3.69% of the Qwen3-30B-A3B checkpoint and 3.01% of the Nemotron-3-Ultra-550B-A55B checkpoint. It thus totals about $1.04\times$ and $1.03\times$ the respective checkpoint. Balanced ownership spreads it across the training ranks: for the 1,121 GB Nemotron-3-Ultra-550B-A55B checkpoint, each of the 32 training ranks of § 8.1 holds about 36 GB.

4.2. Direct projection

In ❷, fixed shard geometry and index mappings let an owner compute the canonical destination of each change locally. For a changed index j of affine task T , the index mapping π_T identifies the canonical destination, and the source projection Q_T produces its updated value:

$$d = \pi_T(j), \quad o_T^{v+1}(j) = I(Q_T(S_T^{v+1}))_d. \quad (1)$$

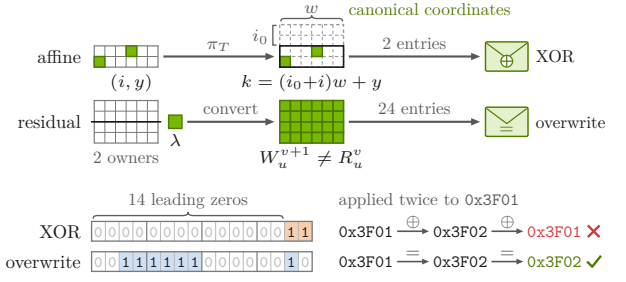


Figure 5 | Top: affine and residual changes. Bottom: XOR and overwrite entries of the example, each applied twice to 0x3F01.

Here, $d = (n, k)$ identifies a canonical tensor name and flat index, and $o_T^{v+1}(j)$ is the updated stored value.

The owner emits each change once for all receivers without knowing their storage offsets. In the example of § 3, a changed local element (i, y) in a row shard that begins at row i_0 maps to canonical destination (n, k) , where $k = (i_0 + i)w + y$ for w columns (Figure 5). Each receiver’s native loader then selects its own column slice.

4.3. Residual conversion

In ❸, residual conversion covers residual tasks, which fall outside direct projection because they lack a supported mapping, can spread a source change across several outputs, or combine inputs from several owners. Examples include stacking, padding, tied weights, adapters, custom postprocessing, and quantization scales. For these tasks, NeMo-DCR constructs the affected canonical tensors and compares their bits with the committed residual baseline (Figure 5).

Change detection covers every input of a conversion because a change in a shared scale factor λ can alter many converted values even when the other source values stay unchanged. For example, a fixed conversion rule may multiply each source value by λ and cast the product. Owners set a flag when a residual task’s shard changes (Figure A.4). Combined with known conversion dependencies, the flags select every canonical tensor affected directly or indirectly. The selected tensors form the required tensor set $\mathcal{U}_{\text{req}}$. The delta omits outputs whose converted bits remain unchanged.

For each selected tensor u , the owners convert their parts, and a canonical-name hash selects the residual-baseline owner that assembles these parts (Figure A.4). The parts must jointly cover every element of u , and overlapping parts must agree bit-for-bit. Comparing the assembled result W_u^{v+1} with its committed residual baseline R_u^v gives

$$D_u = \{(n_u, k) \mid I(W_u^{v+1})_k \neq I(R_u^v)_k\},$$

where n_u is the canonical name. Residual tensors outside $\mathcal{U}_{\text{req}}$ keep their committed values.

Both paths feed one changed set: the emitted destinations combine direct projections $\pi_T(C_T)$ and residual changes D_u . Each emitted destination has a *delta record* with its updated value, and these records form the complete changed set $\mathcal{E}$ that the codec encodes.

5. Mixed XOR/Overwrite Encoding with Compression

In ③, the XOR/overwrite encoding keeps reconstruction exact, and XOR also improves compression. Delta records in canonical coordinates share one codec and one payload format: the affine/residual path decides how a record is built, and the XOR/overwrite encoding decides how it is applied.

Exact encoding rules out arithmetic reconstruction, which can fail to recover the updated stored values: for an old value α and an updated value β , rounding can make $\alpha + (\beta - \alpha)$ differ from β , and such errors can accumulate across versions. NeMo-DCR instead sends either XOR masks of stored values or absolute overwrites. Both reproduce the new bits exactly: a mask flips only the bits that differ, and an overwrite replaces the whole stored value.

For direct projection, the mask of changed index j is

$$x_T^{v+1}(j) = I(S_T^{v+1})_j \oplus I(A_T^v)_j. \quad (2)$$

However, the mask yields the target bits only on a *representation-preserving* path, where the source projection Q_T and native loader preserve dtype and stored bits. In the shared-scale example of § 4.3, multiplication and casting change stored bits, so a source mask does not carry over to the converted values (Figure D.1). Comparison with the residual baseline finds such changes after conversion, and absolute overwrites carry the converted bits.

To exploit unchanged bits within changed values, the default *mixed* mode uses XOR entries (d, x) wherever an affine path is representation-preserving and no other write touches the same bytes. These conditions are decided in advance for each mapping class and loader path (Figure D.1). All other records become overwrite entries (d, o) . In the two MoE checkpoints of § 4, most conversions are affine and preserve stored bits, so XOR covers more than 96% of the weight bytes, as Appendix D details.

XOR entries also compress well: small updates often leave high-order bits unchanged, so XOR masks contain many leading zeros. In the example of § 3, the XOR mask 0x0003 has 14 leading zero bits (Figure 5). Run-length and gap coding shrink location

bytes, and level-1 zstd [2] then compresses both values and locations in each payload.

Yet XOR makes duplicate delivery unsafe: applying a mask twice restores the old bits (Figure 5), so receivers suppress duplicate payloads by their identifiers. For the same reason, retries repair partial writes with overwrites.

6. Recoverable In-Place Refits

In ⑤–⑦, native-loader placement and overwrite recovery allow in-place refits from canonical coordinates without a receiver-side baseline copy. Reimplementing placement would duplicate the rules for fused Q/K/V, routed experts, slicing, padding, transposition, and tied weights. NeMo-DCR instead lets the native loader apply these rules, intercepts the loader’s final storage copies, and specifies loader conditions that guarantee dense-refit bits. Overwrite recovery repairs partial writes, and a joint commit keeps receiver storage and the source baseline in *version alignment*: each transition starts with every required receiver at the committed version.

6.1. Applying deltas through the native loader

Receivers apply payloads through the native loader one *item* at a time, where an item holds one owner’s entries for one tensor. After validating and deduplicating payloads (Figure B.2), each receiver scatters each item into a reusable host *scratch* buffer with the canonical tensor’s shape and dtype. Positions without an entry are inactive and hold a placeholder q (Figure 6, top). The loader resolves names and transforms the scratch, while a PyTorch dispatch hook updates only active positions at the loader’s final storage copies.

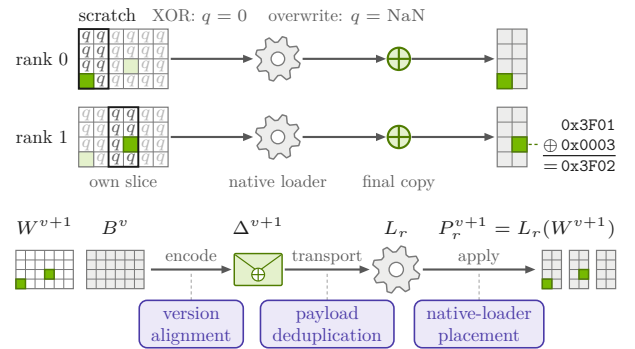


Figure 6 | Top: applying one item in ③ of Figure 4. Bottom: the chain of Proposition 1 and the property used at each arrow.

XOR path. Scratch holds the placeholder $q = 0$ except for masks at changed positions, such as 0x0003 at d in the example. The intercepted copy XORs each

mask into the matching resident element, and the copy’s input must share storage with scratch so the masks arrive unchanged. XOR with $q = 0$ leaves inactive positions unchanged.

Overwrite path. Scratch holds overwrite values and a placeholder that must stay distinct from these values until the first intercepted copy into each destination view. At that copy, non-placeholder positions form the view’s boolean active mask of positions to update (Figure A.10). NaN is the default placeholder.

Receiver memory. Beyond resident weights, receivers keep only the scratch, staged payloads, and temporary buffers for active masks and optional post-apply checks of intercepted copies (Figures C.2 and A.12). A **prewarm** call reserves scratch for the largest canonical tensor and identifies the tensors that the native loader skips on each rank, such as routed experts held by other ranks.

Loader conditions. Exact delta application requires intercepting every storage write: the tensors the loader reports as loaded must match the intercepted copies, and an item without an intercepted copy requires an explicit native skip (Figure A.8). The loader must transform each canonical value independently and deterministically throughout the transition. XOR byte ranges must be disjoint from all other writes, and overlapping overwrites must yield the same bits as a dense refit. NeMo-DCR rejects unsupported loader paths in advance and fails any attempt whose copy breaks these conditions (Table A.2).

6.2. Equivalence to a dense refit

Under these conditions, a NeMo-DCR refit is bit-exact. Let Δ^{v+1} encode $\mathcal{E}$, and let apply_r apply Δ^{v+1} at required receiver r , starting from its storage P_r^v .

Proposition 1 (Dense-refit equivalence). If a first attempt passes every check before its commit and the assumptions of Table A.3 hold, including fixed candidate weights W^{v+1} , committed baseline, source ownership, conversion rules, and loader settings until the commit, then

$$\text{bits}(\text{apply}_r(\Delta^{v+1}, P_r^v)) = \text{bits}(L_r(W^{v+1})) \quad \forall r, \quad (3)$$

where bits reads all rank-local parameter and buffer storage but excludes request state such as the KV cache, which each transition invalidates.

Proof sketch. XOR reconstructs each new value from the resident version- v bits (Figure A.13, top). Overwrite supplies the value itself. Every changed

destination has a record, and destinations without a record keep their version- v bits, which already match a dense refit because their canonical elements did not change. The bottom of Figure 6 outlines the argument, and Appendix A gives the full statement and proof.

6.3. Overwrite recovery and joint commit

Overwrite retries extend Proposition 1 to transitions whose first attempt fails. Partial writes from a failed attempt leave receivers with a mixture of old and updated values. NeMo-DCR stops the failed attempt and waits until its writes can no longer take effect, then recomputes the complete changed set from the same candidate weights and unchanged B^v . Every record becomes an absolute overwrite, supplying the target bits regardless of which earlier writes completed (Figure A.14). The retry thus yields $L_r(W^{v+1})$. Receiver membership changes take effect only between attempts, and new or restarted receivers restore the committed version through a dense refit before joining.

At the end of an attempt, the joint commit publishes the updated policy and binds it to its source baseline in a durable, versioned commit record that a control plane holds and changes only by compare-and-set. The coordinator commits after all required receivers have received every expected payload and finished application, device synchronization, and optional post-apply checks. A commit succeeds only if the commit record still equals the version- v record K^v . Owners then update the task trackers and residual baseline in place (Figure B.3). This update finishes before the transition ends or change detection resumes, so the next delta uses the committed baseline.

The commit record also makes coordinator failover safe: a new coordinator reads the same commit record before deciding whether to retry. Appendix B specifies the protocol as Algorithm B.2, with commit deadlines and the control-plane conditions for failover.

7. Pipelined Delivery without a Cross-Cluster Collective

In ④, pipelined delivery over object storage or a relay tree overlaps other refit stages without a cross-cluster collective. Owners emit the payloads, and both transports share the refit protocol of Appendix B.

7.1. Object-storage and relay-tree delivery

Payload bytes scale with emitted values, locations, and metadata, but transport topology and retries determine how often payloads cross the cluster boundary.

Let V_{payload} be the total bytes in unique compressed payloads before replication to receivers, and V_{cross} the bytes crossing that boundary.

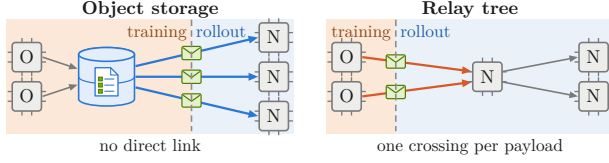


Figure 7 | Logical payload movement per attempt in ❹ of Figure 4, with owners O, rollout nodes N, and the object-storage manifest of expected payloads.

Object storage. This transport separates uploading from downloading (Figure 7, left). Owners upload each payload once and list it in a per-attempt manifest of expected payloads. Rollout nodes fetch every listed payload without a direct link. One stored copy in canonical coordinates serves every rollout node, but each download crosses the cluster boundary, so V_{cross} grows with the number of rollout nodes. A retry uploads the complete changed set again.

Relay tree. Where direct links exist, this transport streams each owner’s payloads to one root in the rollout cluster (Figure 7, right). The root forwards payloads through a balanced tree over local links, and each node’s receivers apply them. Without retransmission, each payload crosses the cluster boundary once per attempt, so $V_{\text{cross}} \approx V_{\text{payload}}$. Appendix B details both transports.

7.2. Overlapping refit stages

Overlap matters because payload volume scales with the changed set, whereas delta construction cost scales with model size: construction scans source shards and converts required residual tensors. To overlap delta construction and delivery, owners group whole tensors into *chunks* for comparison and pack the resulting records into *buckets*, each compressed into one payload (Figure 8, top). Buckets amortize the overhead of metadata, compression, and transfer.

In Figure 8 bottom, owners construct bucket $\ell+1$ while payload ℓ is sent. After `open_apply`, receivers in synchronous RL apply ℓ while $\ell+1$ arrives. Bounded queues slow upstream stages when receivers fall behind, limiting the number of decoded payloads held in memory. Pipeline startup and drain leave stages idle.

A request gate shields requests from partially applied weights and closes at different times in synchronous and asynchronous RL. In synchronous RL, the coordinator closes it when the first attempt starts, and receivers apply payloads during delivery (Figure 8, bottom). In asynchronous RL, receivers stage the

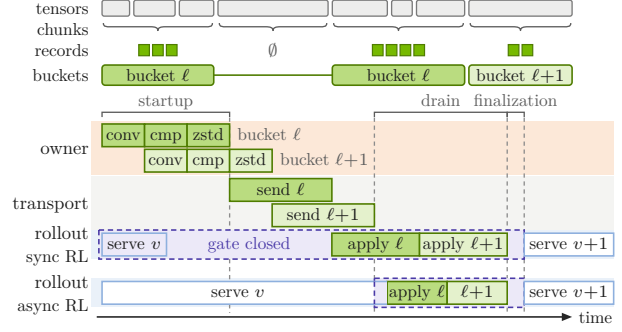


Figure 8 | Top: chunks group whole tensors, and buckets pack their records. An unchanged chunk has none ($\emptyset$) and adds nothing to bucket ℓ , which continues past it. Bottom: stage timeline of residual-task buckets in ❷–❹ of Figure 4 with conversion (conv), comparison (cmp), and the request gate.

compressed delta in host memory while rollout keeps serving version v . After the last payload arrives, the coordinator closes the gate, and receivers apply the delta in place once current requests finish. The gate reopens when the transition ends.

8. Evaluation

Research questions (RQs) 1–3 and 6 test the four contributions, and each subsection names the contributions it tests. RQs 4–5 test their combined effect.

RQ1. How much smaller are XOR masks than overwrite values after compression? (§ 8.2)

RQ2. How do direct projection and XOR encoding change construction time and bytes? (§ 8.3)

RQ3. Are refits bit-exact, and does training under kills follow dense NCCL? (§ 8.4)

RQ4. How much faster is a NeMo-DCR refit than the reference at 120B? (§ 8.5)

RQ5. How do latency and speedup change as models grow from 30B to 1T? (§ 8.6)

RQ6. How much of refit latency do relay-tree transport and delta construction occupy? (§ 8.7)

8.1. Experimental setup and bit-exactness

Testbed. The latency testbed has 32 NVIDIA GB300 GPUs on eight training nodes in one AWS region and 64 H100 GPUs on eight rollout nodes in another. The object-storage transport uses Amazon S3. Each node’s cross-cluster flow was measured at up to 5 Gbps. The latency runs use the BF16 Nemotron and Qwen3-235B-A22B checkpoints of Table C.1, with sizes from 63.2 to 2,242 GB, where 1 GB is 10^9 bytes. The 1T checkpoint doubles the layers of Nemotron-3-Ultra-550B-A55B. We run five warm-up refits and then ten timed refits for each training-side

configuration of parallel layout and replica count. The ten timings of each configuration span less than 1 s from minimum to maximum. We average them, then average these means with equal weight.

Refit settings. Both transports use direct projection and the same mixed-mode codec and bounded-queue policy, with the chunk, bucket, and queue sizes of Appendix C. The latency runs use synchronous RL, the training runs use asynchronous RL, and neither uses model-specific logic.

Element change rates. Latency and delta construction runs use 3% or 5% element change rates as stress cases, since both exceed every rate in Figure 2. Changed source elements are chosen uniformly from each owner’s shards and updated with a relative amplitude of about 10^{-2} . Replicated copies receive the same changes.

Timed window. Refit latency covers comparison and projection, residual conversion, encoding and zstd, transport, receiver staging, loader application, device synchronization, ACK, joint commit, and the in-place baseline update (Figure C.5). It excludes benchmark change injection, post-apply checks, and one-time work such as model loading, initial baseline construction, and receiver prewarm.

Reference. The two clusters share no InfiniBand or Elastic Fabric Adapter (EFA) route for NCCL. Of approaches A–D in Table 1, only full-checkpoint transfer meets both G1 and G4. We therefore use a transport-only full-checkpoint reference: each training node uploads an eighth of the checkpoint to shared object storage, and each rollout node then downloads all of it (Figure C.5). The reference excludes checkpoint save and load, to its advantage, whereas NeMo-DCR refit latency includes every stage of the timed window. Speedup is reference time divided by refit latency.

Bit-exactness. Comparisons with dense refits from the same candidate weights confirmed bitwise equality of every parameter and buffer element, including unchanged elements, in the evaluated BF16 receiver configurations. Every measured NeMo-DCR refit also passed its post-apply checks over all written parameter and buffer bits.

8.2. XOR masks are 1.7–2.2× smaller

XOR masks are 1.7–2.2× smaller than overwrite values after zstd level-1 compression (contribution 2, Table 2). We generate 20 million standard-normal values directly in BF16 and add independent Gaussian noise with standard deviation $a \in \{10^{-4}, 10^{-3}, 10^{-2}\}$ in FP32. We round each result to the nearest BF16 value, as BF16 training does. The comparison isolates

value encoding: XOR and overwrite streams encode the same changed words in input order, without locations or metadata, as Appendix C details.

Table 2 | Compressed changed-value sizes for 20 million synthetic BF16 inputs, where 1 MB is 10^6 bytes.

Noise standard deviation a	Overwrite ↓ (MB)	XOR ↓ (MB)	Reduction ↑
10^{-4}	0.74	0.33	$2.21\times$
10^{-3}	7.02	3.23	$2.18\times$
10^{-2}	25.70	14.93	$1.72\times$

8.3. Projection saves time, XOR saves bytes

Direct projection makes Qwen3 delta construction 1.08–1.16× faster than full conversion, and XOR encoding cuts payload bytes by 38–40% versus overwrites only (contributions 1 and 2, Table 3). We compare three delta construction modes in the 3% and 5% stress cases on Qwen3-30B-A3B and Qwen3-235B-A22B, which use the same mapping classes. Mode X uses direct projection with mixed XOR/overwrite encoding, P uses direct projection with overwrites only, and F uses full conversion, which assembles and converts full tensors, with overwrites only. P versus F isolates direct projection, and X versus P isolates XOR encoding. Mode X spends 7–13% more construction time than P, yet it is the default because transport dominates relay-tree refit latency (§ 8.7).

8.4. Training follows dense NCCL under receiver kills

With receivers killed mid-refit, both NeMo-DCR transports follow dense NCCL’s mean reward and KL trajectories (contribution 3, Figure 9). To exercise in-place application and overwrite retries on receivers that survive a kill, we train Qwen3-30B-A3B on agentic tool calls for 50 GRPO steps with the settings of Appendix E. Training and rollout each use four H100 nodes on a second testbed. We compare three methods: a dense NCCL refit that sends every weight, object-storage NeMo-DCR, and relay-tree NeMo-DCR. These runs use real training updates, which change 1.2% of elements per step on average. KL is the estimated per-token KL divergence between rollout and training policies.

Every five steps in each NeMo-DCR run, we pick a vLLM instance uniformly at random, kill it mid-refit, and restart it five steps later (Figure E.1). Each kill fails the in-flight attempt, and the overwrite retry repairs any partial writes on the other receivers. Restarted receivers restore the committed version before rejoining. We run each method five times, and each NeMo-DCR run uses independent random

Table 3 | Qwen3 delta construction time and payload size before replication. An arrow names the only difference between its modes.

		direct projection			full conversion	
		mixed XOR/overwrite	X	↓ XOR	P	F
		overwrites only		projection		
Qwen3	Change rate	X (s)	P (s)	F (s)	X (GB)	P=F (GB)
30B	3%	2.96	2.77	3.20	1.49	2.44
30B	5%	3.23	2.99	3.29	2.29	3.80
235B	3%	20.23	18.38	19.87	11.37	18.42
235B	5%	23.07	20.42	21.96	17.58	29.36

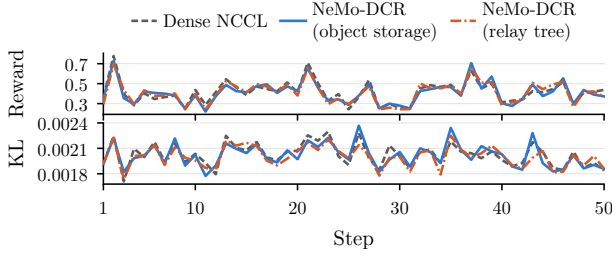


Figure 9 | Qwen3-30B-A3B reward and per-token KL, averaged over five runs. Mid-refit receiver kills occur only in NeMo-DCR runs.

choices. In these runs, every NeMo-DCR refit and overwrite retry passed all of its configured post-apply checks. Over the 50 steps, mean reward is 0.41 for dense NCCL and both NeMo-DCR transports.

8.5. 15–33× faster refits at 120B

Including delta construction and loader application, 120B refits on both transports take 22.6–49.7 s and are 15.1–33.2× faster than the reference (Table 4). Before replication, compressed payloads including locations total 2.4–3.8% of the 247.2 GB checkpoint, 21–24% less than the uncompressed changed values alone, about 3% or 5% of the checkpoint.

Table 4 | Mean 120B refit latency versus the transport-only full-checkpoint reference. Volume is V_{payload} for NeMo-DCR and the checkpoint size for the reference, both measured before replication.

Method	Change rate	Latency ↓	Volume ↓	Speedup ↑
Object storage	3%	24.5 s	5.86 GB	30.6×
Relay tree	3%	22.6 s	5.82 GB	33.2×
Object storage	5%	49.7 s	9.38 GB	15.1×
Relay tree	5%	41.6 s	9.35 GB	18.0×
Reference	N/A	750 s	247.2 GB	1×

8.6. 12–40× faster from 30B to 1T

With either transport, refits of 30B–1T models stay 12–40× faster than the reference in the 3% and 5%

stress cases, although refit latency grows with checkpoint size, from 8.3–14.5 s at 30B to 150–300 s at 1T (Figure 10). At 5%, the relay tree beats object storage at every size, and its advantage grows from 1.13× at 30B to 1.36× at 1T.

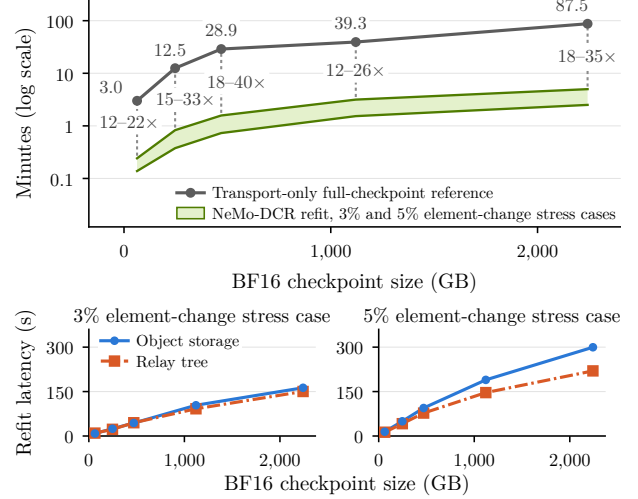


Figure 10 | Top: reference time and mean NeMo-DCR refit latency. The band spans both transports and change rates, labels give reference minutes, and speedups sit on the dotted lines. Bottom: mean refit latency of each transport.

8.7. Relay-tree transport dominates latency

Nsight Systems traces show that relay-tree transport dominates refit latency and overlaps delta construction (contribution 4, Figure 11). The *construction lower bound* covers comparison, projection, residual conversion, encoding, and zstd. The *transport lower bound* covers intervals from payload submission until the sending endpoint receives the rollout node’s queueing acknowledgment. Each lower bound is the time its intervals cover on the busiest owner or sending endpoint, counting overlaps once (Figure C.5). No transport interval includes the staging or application of its own payload.

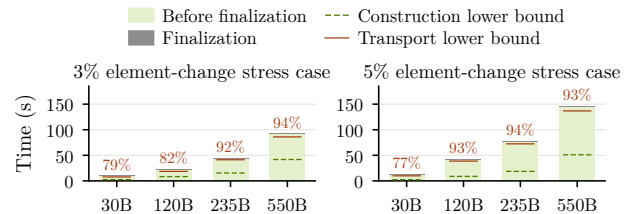


Figure 11 | Relay-tree refit latency and lower bounds. The percentages give the transport lower bound’s share of mean refit latency.

In the stress cases, the transport lower bound accounts for 77–94% of refit latency, versus 18–45% for the

construction lower bound. Pipelined delivery lets delta construction overlap relay-tree transport.

Neither lower bound covers finalization (Figure 8, bottom), which takes 0.2–1.7 s after application and covers device synchronization, ACK, the joint commit, and the in-place baseline update. Refit latency exceeds the larger bound plus finalization by 1.8–9.2 s, spent in startup, drain, and stage handoffs.

9. Conclusion

Because only about 1% of BF16 values change per step, a delta refit can be much cheaper than a full-checkpoint transfer if it addresses placement, exactness, and efficiency, which NeMo-DCR does in canonical coordinates through four contributions. Direct projection avoids full-tensor assembly and conversion for affine changes, residual conversion covers the rest, and mixed XOR/overwrite encoding cuts Qwen3 payload bytes by 38–40% versus overwrites. Recoverable in-place refits match dense refits bitwise without a receiver-side baseline copy, and both transports follow dense NCCL’s mean reward and KL trajectories despite receiver kills. The source baseline stays in host memory and totals about one checkpoint across training ranks. Pipelined delivery without a cross-cluster collective overlaps delta construction with relay-tree transport, and the transport lower bound accounts for 77–94% of refit latency.

In the cross-cluster 3% and 5% stress cases for 30B–1T models, refits are 12–40× faster than the transport-only full-checkpoint reference, and the relay tree takes 150 s instead of 87.5 min at 1T and 3%. These results make per-update refits practical for cross-cluster agentic RL at trillion-parameter scale. NeMo-DCR is open source as part of NeMo RL [22].

References

- [1] J.-P. Aumasson, S. Neves, Z. Wilcox-O’Hearn, and C. Winnerlein. BLAKE2: Simpler, smaller, fast as MD5. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013. doi: 10.1007/978-3-642-38980-1_8.
- [2] Y. Collet and M. S. Kucherawy. Zstandard compression and the ‘application/zstd’ media type. RFC 8878, Internet Engineering Task Force, 2021. URL <https://www.rfc-editor.org/rfc/rfc8878>.
- [3] Cursor Research Team. Composer 2 technical report. *arXiv preprint arXiv:2603.24477*, 2026.
- [4] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram. Check-N-Run: A checkpointing system for training deep learning recommendation models. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*, pages 929–943. USENIX Association, 2022.
- [5] Fireworks AI. Frontier RL is cheaper than you think. [Blog post, archived copy](#), 2026. Accessed August 2026.
- [6] W. Gao, Y. Zhao, D. Muhtar, D. An, X. Shang, T. Wu, L. Cao, S. Xiong, W. Wang, J. Huang, T. Ma, S. Yang, J. Wang, L. Qu, B. Zheng, and W. Wang. ROSE: Rollout on serving GPUs via cooperative elasticity for agentic RL. *arXiv preprint arXiv:2605.06534*, 2026.
- [7] W. Gao, Y. Zhao, T. Wu, S. Xiong, W. Wang, D. An, L. Cao, D. Muhtar, Z. Liu, H. Zhao, J. Huang, S. Yang, Y. Li, W. Su, J. Wang, L. Qu, B. Zheng, and W. Wang. RollArt: Disaggregated multi-task agentic RL training at scale. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation*, pages 863–881. USENIX Association, 2026.
- [8] M. Hershcovitch, A. Wood, L. Choshen, G. Girmonsky, R. Leibovitz, O. Ozeri, I. Ennmouri, M. Malka, P. Chin, S. Sundararaman, and D. Harnik. ZipNN: Lossless compression for AI models. In *Proceedings of the 18th IEEE International Conference on Cloud Computing*, pages 186–198. IEEE, 2025. doi: 10.1109/CLOUD67622.2025.00028.
- [9] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-rank adaptation of large language models. In *Proceedings of the 10th International Conference on Learning Representations*, 2022. URL <https://openreview.net/forum?id=nZeVKeeFYf9>.
- [10] J. Hu, X. Wu, W. Shen, J. K. Liu, W. Wang, S. Jiang, H. Wang, H. Chen, B. Chen, W. Fang, Xianyu, Y. Cao, H. Xu, and Y. Liu. OpenRLHF: A Ray-based easy-to-use, scalable and high-performance RLHF framework. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 656–666. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-demos.48.
- [11] L. Hu, R. Zhao, I. Zhu, Z. Zhang, H. Zhang, H. Yin, and J. Zhao. SparseRL-Sync: Lossless weight synchronization with ~100× less communication. *arXiv preprint arXiv:2605.07330*, 2026.
- [12] Hugging Face. Shipping a trillion parameters with a Hub bucket: Delta weight sync in TRL. [Blog post](#), 2026. Accessed September 2026.

- [13] S. Jiang, T. Shi, S. Zhang, Z. Wang, M. Di Francesco, and B. Zhao. Nereus: Adaptive parallelism for LLM post-training. *arXiv preprint arXiv:2609.34645*, 2026.
- [14] Kimi Team. Kimi K2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.
- [15] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626. ACM, 2023. doi: 10.1145/3600006.3613165.
- [16] W. Li, X. Chen, H. Shu, Y. Tang, and Y. Wang. ExCP: Extreme LLM checkpoint compression via weight-momentum joint shrinking. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 27575–27588, 2024. URL <https://proceedings.mlr.press/v235/li24m.html>.
- [17] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *Proceedings of the 6th International Conference on Learning Representations*, 2018. URL <https://openreview.net/forum?id=SkhQHMWOW>.
- [18] J. Liu, G. Xiao, K. Li, J. D. Lee, S. Han, T. Dao, and T. Cai. BitDelta: Your fine-tune may only be worth one bit. In *Advances in Neural Information Processing Systems 37*, pages 13579–13600. Curran Associates, 2024. doi: 10.52202/079017-0434.
- [19] E. Miah and E. Belilovsky. Understanding and exploiting weight update sparsity for communication-efficient distributed RL. *arXiv preprint arXiv:2602.03839v2*, 2026.
- [20] Moonshot AI. Checkpoint Engine: A simple middleware to update model weights in LLM inference engines. [GitHub repository, commit d1de07b3](#), 2026. Accessed August 2026.
- [21] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. ACM, 2021. doi: 10.1145/3458817.3476209.
- [22] NVIDIA. NeMo RL: A scalable and efficient post-training library. [GitHub repository, commit 72149d09](#), 2026. Accessed August 2026.
- [23] NVIDIA. NCCL-Reshard refit. [NeMo RL pull request #2971](#), 2026. Accessed August 2026.
- [24] NVIDIA. Two-stage SWE RL for Qwen3-30B-A3B-Thinking. [NeMo RL guide, commit b7a4d95d](#), 2026. Accessed September 2026.
- [25] NVIDIA. NeMo Megatron Bridge: Training library for Megatron-based models with bidirectional Hugging Face conversion capability. [GitHub repository, commit 554c7b93](#), 2026. Accessed August 2026.
- [26] Y. Peng, J. Zhang, W. Zhou, R. Xu, R. Yan, W. Dong, Y. Gao, Z. Ding, T. Yang, and B. Yuan. AReaL-DTE: Sparse policy-weight transfer for online agentic reinforcement learning. *arXiv preprint arXiv:2608.00455*, 2026.
- [27] Prime Intellect Team, S. Jaghouar, J. Mattern, J. M. Ong, J. Straube, M. Basra, A. Pazdera, K. Thaman, M. Di Ferrante, F. Gabriel, F. Obeid, K. Erdem, M. Keiblinger, and J. Hagemann. INTELLECT-2: A reasoning model trained through globally decentralized reinforcement learning. *arXiv preprint arXiv:2505.07291*, 2025.
- [28] P. Qi, Z. Liu, X. Zhou, T. Pang, C. Du, W. S. Lee, and M. Lin. Defeating the training-inference mismatch via FP16. *arXiv preprint arXiv:2510.26788*, 2025.
- [29] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020. doi: 10.1109/SC41405.2020.00024.
- [30] C. Ruan, G. Luo, X. Wan, L. Zhao, Q. Wang, J. Zhu, D. Xu, G. Xu, D. Wei, X. Liu, C. Li, H. Sun, L. Luo, C. Miao, and J. Li. AuroraRL: Fast, fault-tolerant, and cost-efficient reinforcement learning over decentralized network. *arXiv preprint arXiv:2602.11456*, 2026.
- [31] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu, and D. Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
- [32] G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the 20th European Conference on Computer Systems*, pages 1279–1297. ACM, 2025. doi: 10.1145/3689031.3696075.
- [33] THUDM. Delta weight sync. [slime documentation, commit 474861aa](#), 2026. Accessed September 2026.
- [34] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, 1996. URL https://rsync.samba.org/tech_report/.

- [35] verl. EP-aware sharded delta export (fused expert stacks). [GitHub pull request #7085](#), 2026. Accessed September 2026.
- [36] verl. Sharded delta weight sync over NCCL for disaggregated rollout. [GitHub pull request #6974](#), 2026. Accessed September 2026.
- [37] vLLM. Add sparse NCCL weight transfer support for in-place updates. [GitHub pull request #40096](#), 2026. Accessed September 2026.
- [38] T. Vogels, S. P. Karimireddy, and M. Jaggi. PowerSGD: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems 32*, pages 14259–14268. Curran Associates, 2019.
- [39] X. Yao, Q. Hu, and A. Klimovic. DeltaZip: Efficient serving of multiple full-model-tuned LLMs. In *Proceedings of the 20th European Conference on Computer Systems*, pages 110–127. ACM, 2025. doi: 10.1145/3689031.3717468.
- [40] C. Ye, H. Zhang, M. Han, B. Zhong, X. Li, Q. Chen, X. Zhang, W. Zhang, K. Jiang, W. Zhang, H. Sun, W. Xiao, A. C. Arpacı-Dusseau, and R. H. Arpacı-Dusseau. TensorHub: Scalable and elastic weight transfer for LLM RL training. In *Proceedings of the 32nd Symposium on Operating Systems Principles*, pages 882–897. ACM, 2026. doi: 10.1145/3830418.3843859.
- [41] Y. Zhong, Z. Zhang, X. Song, H. Hu, C. Jin, B. Wu, N. Chen, Y. Chen, Y. Zhou, C. Wan, H. Zhou, Y. Jiang, Y. Zhu, and D. Jiang. StreamRL: Scalable, heterogeneous, and elastic RL for LLMs with disaggregated stream generation. *arXiv preprint arXiv:2504.15930*, 2025.

Appendices

Appendix A states the source and native-loader conditions and proves the dense-refit equivalence. Appendix B specifies the refit protocol and its failure handling. Appendix C gives integration and measurement details, Appendix D the affine and XOR coverage and the payload format, Appendix E the GRPO settings, and Appendix F the evidence behind Table 1. Table A.1 summarizes the main notation, and Figure A.1 places it on the refit path.

A. Dense-Refit Equivalence

This appendix states the conditions of Proposition 1 and proves it. Appendix A.1 also lists the direct-projection mapping classes of § 4.

Table A.1 | Main notation. Other symbols are defined in the sections where they are used.

Symbol	Meaning	Where
W^v	Canonical tensors at version v , including versioned buffers	2.1
L_r, P_r^v	Receiver r ’s native loader and rank-local storage $L_r(W^v)$	2.1
B^v	Committed source baseline: task trackers and residual baseline	4.1
J_T, C_T	Owned and changed source indices of task T	4.1
S_T^{v+1}, A_T^v	Candidate shard and tracker of task T	4.1
$I(X)_j$	Stored bit pattern of X at index j	4.1
π_T, Q_T	Index mapping and source projection of an affine task	4.2
$d = (n, k)$	Canonical destination: tensor name and flat index	4.2
R_u^v, D_u	Residual baseline and changed destinations of tensor u	4.3
$\mathcal{U}_{\text{req}}$	Required tensor set of an attempt	4.3
$\mathcal{E}$	Complete changed set of delta records	4.3
$(d, x), (d, o)$	XOR entry and overwrite entry	5
q	Placeholder at inactive scratch positions	6.1
Δ^{v+1}	Encoding of $\mathcal{E}$ applied at each receiver	6.2
apply_r , bits	Delta application at receiver r and stored bits of rank-local parameters and buffers	6.2
K^v	Commit record naming version v and its baseline	6.3
V_{payload}	Compressed payload bytes before replication	7.1
V_{cross}	Bytes crossing the cluster boundary	7.1
e	Delta record: destination, overwrite value, XOR mask, and path tag	A.1
$\mathcal{T}_{\text{aff}}, \mathcal{T}_{\text{res}}, \mathcal{U}$	Affine tasks, residual tasks, and residual-baseline tensors	A.1
$g, \mathcal{M}$	Candidate-baseline identifier and integration setup	A.1
$\rho_r(e)$	Destination byte ranges written on receiver r for record e	A.2
τ	Attempt identifier, returned by prepare	B
m	Encoding mode: mixed or overwrite	B

A.1. Source conditions and version state

This subsection states the source-side conditions and version state of Proposition 1: each unique source element has one owner and one conversion task, the

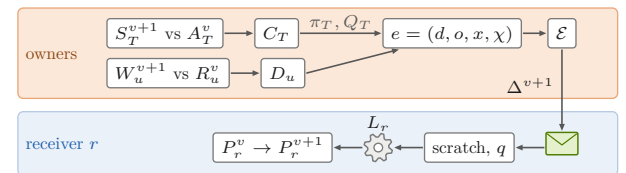


Figure A.1 | Main notation of Table A.1 on the refit path. Owners compare the candidate shard S_T^{v+1} with its tracker A_T^v , or a converted tensor W_u^{v+1} with its residual baseline R_u^v , and emit records e into $\mathcal{E}$, whose encoding Δ^{v+1} each receiver applies through scratch and its native loader L_r .

committed baseline stays fixed until the joint commit, and version alignment keeps every required receiver at the committed version.

Ownership. Replication adds copies but not owners. The parallel layout identifies replica groups, which include the data-, context-, and expert-data-parallel replicas and the tensor-parallel copies of replicated parameters. Every training step keeps the copies within each replica group bitwise identical, so a stable name hash selects one owner among the copies of each shard (Figure A.2). Tied weights likewise have one owner and one task.

Change detection. As in § 4.1, owners compare stored bit patterns, not floating-point values. Floating-point equality treats $+0.0$ and -0.0 as equal although their BF16 patterns $0x0000$ and $0x8000$ differ. A zero that changes sign would then get no record, and receivers would keep the old sign bit. The `bitdiff` call of Table B.1 compares the stored patterns, so it detects this change (Figure A.2).

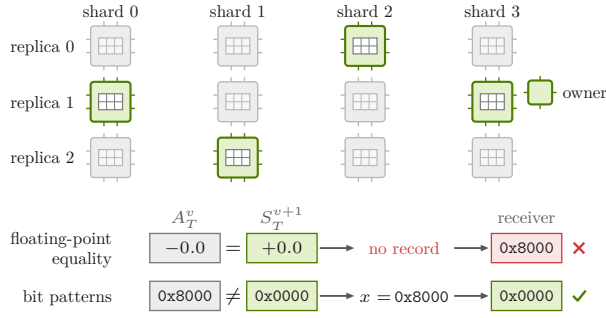


Figure A.2 | Top: ownership across data-parallel replicas. Each replica holds a copy of every shard, and a stable name hash picks one owner per shard and spreads the owners across the replicas, so each unique source element is compared once. Bottom: change detection at a signed zero. Floating-point equality treats -0.0 and $+0.0$ as equal, so the change gets no record and the receiver keeps $0x8000$. Comparing bit patterns finds the change, and the XOR mask $0x8000$ flips the receiver's sign bit, which gives $0x0000$, the value a dense refit writes.

Affine tasks. The mapping classes Direct, Replicated, ColumnParallel, RowParallel, and GatedMLP of Megatron Bridge determine canonical coordinates from local indices, shard geometry, parallel ranks, and global expert indices [25]. Figure A.3 shows each class for two tensor-parallel ranks. For direct projection, the Megatron Bridge integration supports only these affine mapping classes and Auto mappings that

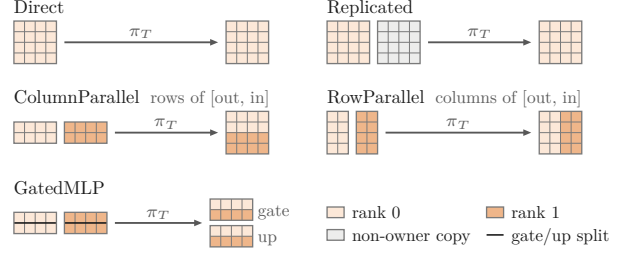


Figure A.3 | Affine mapping classes for two tensor-parallel ranks. Each rank's shard maps through π_T to fixed canonical coordinates, and only the owner's copy of a replicated tensor is projected.

permute no dimensions and act as ColumnParallel, RowParallel, or Replicated mappings. All of them use Bridge metadata for shard axes, offsets, and gate/up splitting. A custom export hook postprocesses converted tensors, so it disables direct projection.

Residual tasks. Change flags select the residual tensors that each attempt converts. Tied embeddings, the output head that depends on them, and any source tensor that also feeds a residual conversion always use residual tasks. For residual tasks, the training cluster's `all_reduce_flags` collective combines the change flags of § 4.3 by task identifier over all owners, and each owner contributes zero for tasks it does not hold (Figure A.4). Before assembly, the flags and conversion dependencies fix the required tensor set $\mathcal{U}_{\text{req}} \subseteq \mathcal{U}$ for the attempt, where $\mathcal{U}$ indexes the residual-baseline tensors. A custom export hook leaves conversion dependencies unknown, so any flagged task selects every residual tensor.

Residual assembly. Residual tasks whose outputs form one canonical tensor are converted together when any of their shards changes. For example, per-expert tasks are converted together when the checkpoint stores all experts as one stacked tensor. A canonical-name hash selects the residual-baseline owner of each residual tensor u from the fixed owners. Each owner's part of u carries the tensor name, shape, dtype, and canonical indices it covers. Figure A.4 shows one such assembly, and Figure 5 contrasts this path with direct projection.

Records. The affine and residual paths both emit records $e = (d, o, x, \chi)$ in $\mathcal{E}$ (Figure A.5), with destination d , overwrite value o , XOR mask x , and path tag $\chi \in \{\text{affine}, \text{residual}\}$. For residual records, $x = \perp$.

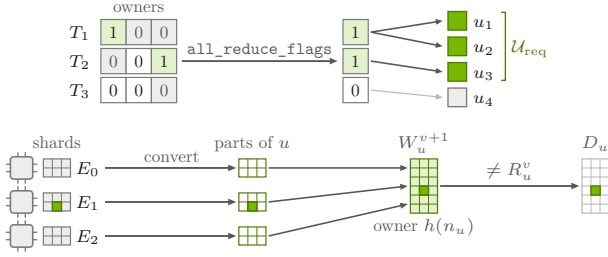


Figure A.4 | Top: residual change flags. Each owner sets 1 for its residual tasks whose shards changed, 0 for its other tasks, and a gray 0 for tasks it does not hold. The combined flags and the conversion dependencies select $\mathcal{U}_{\text{req}}$. Bottom: residual assembly of a stacked expert tensor u . One changed element in expert E_1 makes every owner convert its part. The owner selected by a hash h of the canonical name n_u assembles W_u^{v+1} , and the comparison with R_u^v puts only the changed element into the set D_u .

	d	o	x	χ
$e =$	(n, k)	0x3F02	0x0003	affine
$e =$	(n_u, k)	0x4080	$\perp$	residual

Figure A.5 | Delta records, one per path. The affine record of the example carries the destination, the overwrite value 0x3F02, and the XOR mask 0x0003, while a residual record has no XOR mask ($x = \perp$) and carries only its overwrite value.

Version state. For affine tasks $\mathcal{T}_{\text{aff}}$ and residual tasks $\mathcal{T}_{\text{res}}$, the committed source baseline is

$$B^v = \{A_T^v\}_{T \in \mathcal{T}_{\text{aff}} \cup \mathcal{T}_{\text{res}}} \cup \{R_u^v\}_{u \in \mathcal{U}}.$$

The baseline B^v reconstructs W^v : the source projections Q_T map the affine-task trackers into canonical coordinates, and the residual-baseline tensors supply the rest. The authoritative commit record $K^v = (v, g_v)$ names version v and the identifier g_v of its baseline B^v . With W^{v+1} and B^v fixed, the **prepare** call of Table B.1 binds a fresh candidate-baseline identifier g to the attempt. The logical candidate baseline $B(g) = \{A_T^g\} \cup \{R_u^g\}$ is defined by $A_T^g = S_T^{v+1}$ for every task and $R_u^g = W_u^{v+1}$ for every required residual tensor, with $R_u^g = R_u^v$ for every other residual tensor. Likewise, $B(g)$ reconstructs W^{v+1} when every changed residual tensor is in $\mathcal{U}_{\text{req}}$, as Figure A.6 shows for both baselines.

Fixed state. The candidate weights W^{v+1} , source ownership and membership, and integration setup $\mathcal{M}$ remain fixed across the attempts of the transition, and B^v remains fixed until commit. The integration setup $\mathcal{M}$ contains the conversion rules with their mapping and dependency metadata, the native loaders with their settings and XOR/overwrite/skip classi-

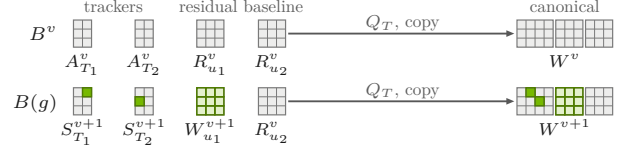


Figure A.6 | Committed and candidate baselines. Each row shows a baseline and the canonical tensors it reconstructs through the projections Q_T and the residual tensors. $B(g)$ replaces every tracker by its candidate shard and the required residual tensor u_1 by its converted value, while u_2 keeps $R_{u_2}^v$.

fications, and the transport adapter of Figure C.1. To keep W^{v+1} fixed, training takes its next optimizer step only after the transition ends (Figure A.7). Source membership is the set of training ranks, and changing it requires redistributing or reinitializing the baseline before another transition.

Version alignment. The protocol preserves *version alignment* between the source baseline and receiver storage: when a transition starts, every required receiver holds P_r^v for the version v that the commit record names. Receivers obtain P_r^v from the committed transition to version v or from a dense refit, which also restores new or restarted receivers. Resident weights change only through refits. To keep this alignment, a commit requires completion acknowledgments from every required receiver, and a new or restarted receiver joins only after restoring the committed version. Figure A.7 follows one transition, and Appendix B.2 specifies both rules.

A.2. Native-loader conditions

For each source and loader path, Table A.2 lists the operations it allows, the rules the integration enforces, and the required assumptions. Loader paths whose required assumptions do not hold are rejected in advance, and a copy that breaks an integration rule at run time fails the attempt.

Native skips. The **prewarm** call runs the loader with tensor metadata to discover which tensors the loader skips on each receiver rank. Any reported load without an intercepted copy fails the attempt, as Figure A.8 shows.

Loader properties. Beyond the conditions of § 6.1, the integration must establish two loader properties: scratch bits remain unchanged before an XOR copy, and model parameters and buffers keep their existing storage (Figure A.9). Because runtime checks at intercepted copies cannot establish

Table A.2 | Operations, integration rules, and assumptions for the default BF16 source and loader paths.

Path	Operations	Integration rule	Required assumptions
<i>Source</i>			
Affine	Direct, Replicated, ColumnParallel, RowParallel, GatedMLP, and qualifying Auto	Supported mapping classes; no custom export hook	Fixed deterministic Bridge mappings; stored bits preserved for XOR entries
Residual	Other mapping classes, stacking, padding, tied weights, adapters, and custom postprocessing	Residual conversion; forced overwrite	Complete dependency metadata, or every residual tensor selected when unknown
<i>Loader</i>			
XOR	Identity, views, slices, splits, fusions	Copy input shares scratch storage; same dtype; no within-item overlap	Stored bits preserved; no overlap across items
Overwrite	Identity, views, slices, splits, fusions, casts, pointwise transforms such as $-\exp(\cdot)$	Copy input is floating-point; non-NaN active values; active mask remains valid	Placeholders preserved until each view's first copy; each output depends only on its input
Native skip	Rank-local skips	Explicit skip report; reported loads match intercepted copies	Accurate reports; interception of every storage write

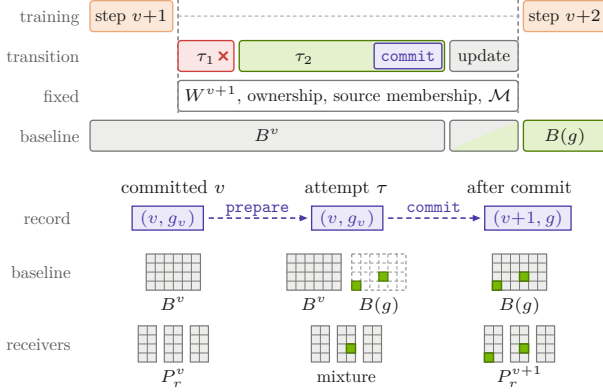


Figure A.7 | Top: what stays fixed during a transition. Training takes its next optimizer step only after the transition ends. The candidate weights W^{v+1} , source ownership and membership, and the integration setup $\mathcal{M}$ stay fixed across the failed attempt τ_1 and its retry τ_2 , and B^v stays fixed until the commit. Bottom: version state at three points of the transition. During an attempt, the commit record and B^v stay fixed, the dashed candidate baseline $B(g)$ exists only logically, and receivers may hold a mixture of old and updated values. After the commit, owners update the baseline in place to $B(g)$.

these properties alone, the integration also validates the configured loader in advance.

XOR copies. The loader must preserve every bit pattern an XOR mask can contain at its dtype width, including NaN, infinity, subnormal, and signed-zero patterns (Figure A.9). Operations such as `nan_to_num`, clipping, and casts therefore use overwrite only. An XOR copy fails the attempt if its

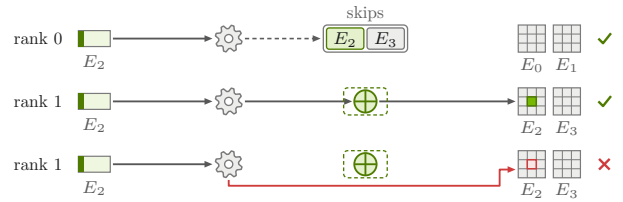


Figure A.8 | Native skips for one routed-expert item E_2 , which every receiver gets. The loader on rank 0 skips E_2 , as `prewarm` recorded, and writes nothing. On rank 1 it writes E_2 through the final storage copy inside the dashed dispatch hook. In the bottom lane, a write bypasses the hook, so a reported load has no intercepted copy and the attempt fails.

input no longer shares scratch storage, its input and destination dtypes differ, or its destination byte range overlaps another copy within the item.

Overwrite copies. The default overwrite paths require a floating-point copy input, whose non-NaN positions form each destination view's active mask. An active value indistinguishable from the placeholder fails the attempt before the item is written. Active values are cast to the destination dtype. Later intercepted writes to the same destination view by a pointwise loader transform may reuse that active mask. The ordered writes of that transform must preserve the active mask and produce the same final bits as a dense refit (Figure A.10). Transforms that erase the placeholder require another validated placeholder scheme satisfying the conditions below.

Placeholder schemes. To validate a placeholder scheme for an overwrite path, the integration traces

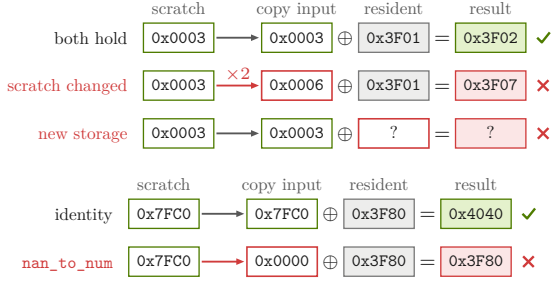


Figure A.9 | Top: loader properties on the XOR path of the example. With unchanged scratch bits and existing parameter storage, the intercepted copy XORs the mask into the resident version- v bits and yields 0x3F02. An in-place loader operation on scratch alters the mask, so both give wrong bits. Bottom: XOR masks on two loader paths. The mask between the resident 1.0 (0x3F80) and the updated 3.0 (0x4040) is the BF16 NaN pattern 0x7FC0. An identity path keeps the mask and yields 3.0, whereas `nan_to_num` turns the mask into 0x0000 and leaves 1.0 unchanged.

placeholders and active values through `nan_to_num`, clipping, casts, quantization, and other loader operations. Outputs must be deterministic and depend only on their corresponding inputs across supported shapes, dtypes, and value ranges, under fixed loader settings and cast behavior. The transformed placeholder must remain distinct from all active values until the active mask is formed. Overwrite retries keep the integration setup $\mathcal{M}$ fixed but may choose another of the integration’s validated placeholder schemes. Figure A.10 contrasts a cast with `nan_to_num`.

Write overlap. Let $\rho_r(e)$ denote the set of destination byte ranges written on receiver r when applying the entry encoded from record e . The overlap rules of § 6.1 apply to these ranges across entries, items, payloads, and loader calls. Runtime checks for XOR overlap are reset at each item, so the integration validates in advance that XOR writes do not overlap writes from other items. A loader change requires another `prewarm` call and revalidation of these rules and the loader properties. Figure A.11 shows $\rho_r(e)$ on a fused path.

Post-apply checks. For an XOR copy, the optional post-apply check saves the destination’s pre-copy bits and verifies that XORing the mask into the updated bits restores the pre-copy bits (Figure A.12). For an overwrite copy, the check compares the written bits with the overwrite values transformed by the native loader.

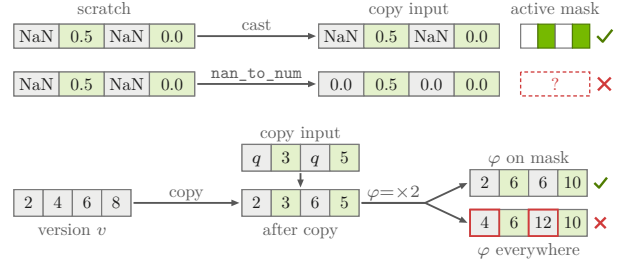


Figure A.10 | Top: the NaN placeholder on two overwrite paths, with placeholders in gray and active values in green. A cast keeps the NaN placeholders, so the non-NaN positions of the copy input form the active mask. `nan_to_num` maps the placeholders to 0.0, which is also an active value, so the mask cannot be formed and the path needs another validated placeholder scheme. Bottom: ordered writes to one destination view. The first intercepted copy forms the active mask from the non-placeholder positions of its input. The intercepted write of a later pointwise transform φ reuses that mask and yields the same bits as a dense refit. Applied everywhere, the doubling φ would transform the inactive positions a second time, because their version- v values are already transformed.

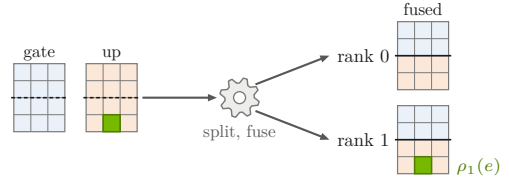


Figure A.11 | A default loader path. The native loader splits the canonical gate and up projections across two tensor-parallel ranks at the dashed lines and fuses them per rank at the solid lines. Each output element has one canonical source, so a changed element of the up projection has one destination range $\rho_1(e)$ on rank 1.

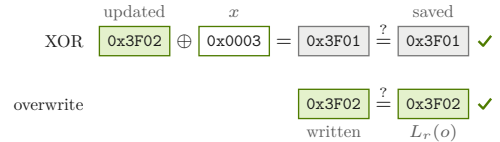


Figure A.12 | Optional post-apply checks on the example. For an XOR copy, XORing the mask into the updated bits must restore the saved pre-copy bits. For an overwrite copy, the written bits must equal the overwrite value o transformed by the native loader.

A.3. Equivalence proof

The argument combines complete change detection and the XOR/overwrite encoding with the three properties of Figure 6: version alignment between the source baseline and receiver storage, pay-

load deduplication at ingestion, and native-loader placement at the receiver.

Full statement of Proposition 1. For a first attempt that passes every check up to and including the `flush` call of Table B.1, receiver storage satisfies (3) under the following assumptions (Table A.3). The candidate weights W^{v+1} , baseline B^v , source ownership and membership, and integration setup $\mathcal{M}$ remain fixed until commit, and $\mathcal{M}$ includes the conversion rules and loader settings. Each required receiver initially holds P_r^v , and receiver membership stays fixed during the attempt. Replicated source copies agree bit-for-bit, and the owners cover each unique source element exactly once. Every required residual tensor is fully assembled, with overlapping parts agreeing bit-for-bit. Projected values and overwrite values are computed deterministically, and every XOR-encoded record uses a representation-preserving path. The residual dependency metadata is complete, or unknown dependencies select every residual tensor, so that W_u^{v+1} equals R_u^v bit-for-bit for every $u \notin \mathcal{U}_{\text{req}}$. The value written to each destination byte range depends only on its corresponding canonical element. The loader classification covers every final storage copy, and the loader conditions of § 6.1 and Appendix A.2 hold.

Table A.3 | Assumptions of Proposition 1, how each is established, and the figures that illustrate it.

Assumption	Established by	Figure
W^{v+1} , B^v , source ownership and membership, and $\mathcal{M}$ fixed until commit	Training waits for the transition; owners update B^v only after commit	A.7
Receivers start from P_r^v ; receiver membership fixed per attempt	Version alignment; receiver membership changes only between attempts	A.7, B.5
Copies agree; one owner per source element	Identical replica updates; name-hash ownership	A.2
Residual tensors fully assembled and consistent	Checks of <code>build_complete</code> in Table B.1	A.4
Deterministic values; XOR only on representation-preserving paths	Fixed conversion rules; encoding decided per mapping class and loader path	D.1
Dependencies complete, or every residual tensor selected	Change flags with dependency metadata	A.4
Each written value depends only on its canonical element	Supported loader paths	A.11
Every final storage copy classified; loader conditions hold	<code>prewarm</code> skip discovery, copy interception, and advance validation	A.8, A.9

XOR invariant. For each XOR-encoded record e from affine task T and source index j , the *XOR invariant* states that every range $\rho \in \rho_r(e)$ holds the matching source-baseline bits until its XOR copy:

$$\text{bits}(P_r^v)|_{\rho} = I(A_T^v)_j, \quad e = (\pi_T(j), o, x, \text{affine}),$$

where each range ρ spans one stored value. Version alignment ensures that the receiver holds these bits: its resident weights are at version v , the baseline version used to compute the XOR mask. Under the proposition’s assumptions, the projection Q_T and loader preserve dtype and stored bits, and no other entry writes to that range, so the invariant holds without transmitting baseline values (Figure A.13).

Proof. Under the assumptions of the proposition, owners compare every unique source element and required residual tensor bitwise, affine outputs depend only on their source elements, and residual tensors outside $\mathcal{U}_{\text{req}}$ keep their committed bits. Hence every changed canonical destination has a record, and the conflict checks leave it exactly one record or several overwrite records with identical value bits. For an XOR-encoded record e from affine task T and source index j , the mask is $x = I(S_T^{v+1})_j \oplus I(A_T^v)_j$. Scratch carries the item’s XOR masks at changed positions and zeros elsewhere. Payload deduplication, together with the arrival check and queue draining of `flush` in Appendix B.1, ensures that the receiver applies every XOR and overwrite entry exactly once. For every $\rho \in \rho_r(e)$, the invariant and the representation-preserving path give $\text{bits}(P_r^v)|_{\rho} \oplus x = \text{bits}(L_r(W^{v+1}))|_{\rho}$.

For overwrite entries, a validated placeholder scheme establishes the active mask, and the loader conditions require the ordered writes of every pointwise loader transform to preserve it. Because each written value depends only on its canonical element, the complete overwrite sequence writes $\text{bits}(L_r(W^{v+1}))$ at active positions and nowhere else.

Every byte range written for an emitted destination therefore holds its target bits, and every remaining byte range keeps its version- v bits (Figure A.13). The canonical element of each remaining range did not change, and the value a dense refit writes there depends only on that element, so these bits already equal $\text{bits}(L_r(W^{v+1}))$. This establishes (3).

Retries. After the `stop_and_wait` call of Table B.1 completes for a failed attempt, a retry that passes every check up to and including `flush` has written every byte range of an emitted destination as an absolute overwrite. Every attempt recomputes $\mathcal{E}$ deterministically from the same W^{v+1} , B^v , ownership,

and $\mathcal{M}$. If every write of the earlier attempts passed the dispatch hook, those attempts wrote only these ranges, and such a retry turns any mixture they left into $\text{bits}(L_r(W^{v+1}))$ before commit (Figure A.14). Retries are therefore safe to repeat. A write that bypasses the hook can reach bytes outside these ranges. Its attempt fails (Figure A.8), as does every retry with that receiver, since the loader repeats the write. After retirement or the operator abort of Appendix B.2, a dense refit restores the receiver.

B. Refit Protocol

This appendix specifies the protocol operations behind the two transports of § 7 and the commit and recovery rules of § 6.3.

The active coordinator runs one transition at a time. Each attempt has a fresh attempt identifier τ and candidate-baseline identifier g , under which the complete changed set $\mathcal{E}$ is streamed. Appendix A.1 defines the committed and candidate baselines and the state

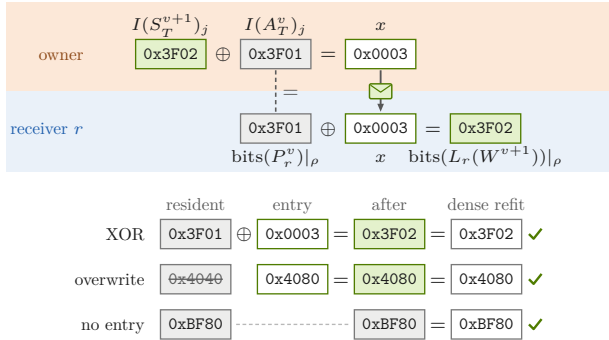


Figure A.13 | Top: XOR invariant for the changed element of the example. Version alignment keeps the receiver's resident bits equal to the owner's baseline bits, marked by the dashed equality, so the owner sends only the mask x across the cluster boundary. Bottom: the three kinds of receiver byte ranges in the proof. An XOR entry turns the resident version- v bits into the target bits, an overwrite entry replaces the resident bits, and a range without an entry keeps its version- v bits, which equal the dense-refit bits because its canonical element did not change.

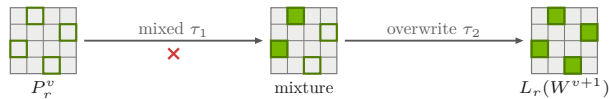


Figure A.14 | Overwrite retry on one receiver, with emitted ranges outlined and updated bits in green. Every attempt emits the same ranges and writes nowhere else. The mixed-mode attempt τ_1 fails after writing two of them, and the overwrite retry τ_2 writes all four, which turns the mixture of old and updated values into the dense-refit result $L_r(W^{v+1})$.

that remains fixed across retries.

In the signatures of Table B.1, $\mathcal{M}$ is the fixed integration setup of Appendix A.1. Mode m is mixed or overwrite, and overwrite mode encodes every record as an absolute overwrite. The symbols p , id , ack , and σ denote a payload, its identifier, an acknowledgment, and a success or failed status, and c is a commit outcome as defined in Appendix B.2. Task T 's change flag is f_T . Inside `build_complete`, `bitdiff` detects changes, owners project affine changes into canonical coordinates, `all_reduce_flags` fixes $\mathcal{U}_{\text{req}}$, and `convert` produces the residual parts.

Failure model. Owners, receivers, transports, and coordinators may crash or lose messages (Table B.2), but they do not act maliciously, and channels are trusted or encrypted. A control plane, assumed available and clock-synchronized with coordinators, holds the durable commit record, accepts `prepare` only from the active coordinator, and raises a `conflict` exception for any later `prepare` or `commit` of a replaced coordinator. An attempt's payloads, control calls, and gate operations carry the attempt identifier τ that its `prepare` returned, and receiver and transport endpoints ignore them once a later `prepare` has replaced that attempt, except that its `stop_and_wait` returns at once. If retries cannot repair a failure, operator intervention is required.

B.1. Admission and delivery

Each attempt runs `prepare`, then streams, checks, applies, and flushes its payloads as in Algorithm B.1.

Admission. The `prepare` call checks the metadata and loader classifications in $\mathcal{M}$, then binds g , mode m , and the set of required receivers to the fresh attempt. The first successful `prepare` admits the transition. A failed `prepare` check raises `reject`, and any other failure raises `failed`. Before admission, a rejection ends the transition. Any other failure before admission retries the first `prepare` in mixed mode. After admission, every `reject` or `failed` exception triggers overwrite recovery, wherever in the attempt the failure occurs. Algorithm B.2 starts in mixed mode, or in overwrite mode when it takes over an admitted transition after coordinator failover (Figure B.1).

Streaming. Receivers validate and deduplicate payloads as they arrive. In synchronous RL, they decode them into the bounded queue but apply them only after the coordinator closes the gate with

Table B.1 | Protocol operations. Operations that take an attempt identifier τ act only on that attempt. Appendix B.2 defines commit outcomes and the recovery rules.

Operation	Effect
<i>Source and coordinator</i>	
<code>refit($W^{v+1}, B^v, K^v, \mathcal{M}$)</code>	Run a transition with overwrite recovery
<code>prepare($v+1, \mathcal{M}, g, m$) $\rightarrow \tau$</code>	Validate $\mathcal{M}$, then bind g, m , and required receivers to a fresh attempt
<code>build_complete($W^{v+1}, B^v, \mathcal{M}, g$) $\rightarrow \{\mathcal{E}_\ell\}_\ell$</code>	Stream changed-set buckets under g and check each for completeness and conflicting canonical values
<code>bitdiff(X, Y)</code>	Compare equal-width stored values and return changed indices
<code>all_reduce_flags($\{(T, f_T)\}) \rightarrow \mathcal{U}_{\text{req}}$</code>	Reduce flags within the training cluster to find all affected tensors
<code>convert(T, u)</code>	Produce task T 's part of canonical tensor u
<code>encode($\mathcal{E}_\ell, m$) $\rightarrow (p_\ell, \sigma_\ell)$</code>	Encode bucket $\mathcal{E}_\ell$ as compressed XOR or overwrite entries
<code>send(τ, p_ℓ)</code>	Send identified payload p_ℓ under τ
<code>wait_for_requests(τ, v)</code>	Close the request gate if open, then drain version- v requests
<code>open_apply(τ)</code>	Let receivers decode staged payloads and apply queued payloads
<code>flush(τ) $\rightarrow (ack, \sigma)$</code>	Check arrivals, drain receiver queues, synchronize devices, report any configured post-apply checks, acknowledge
<code>commit(τ) $\rightarrow c$</code>	Conditionally replace K^v with $(v+1, g)$ after flush
<code>read_commit(τ) $\rightarrow K$</code>	Return the record once no commit of τ can change it
<code>mark_obsolete(τ)</code>	Invalidate τ for later payloads, <code>open_apply</code> , and <code>commit</code>
<code>stop_and_wait(τ) $\rightarrow ack$</code>	Stop delivery, then wait for writes and acknowledgments
<code>cleanup(τ)</code>	Release the payloads and, on failure, the delta records
<i>Receiver and transport endpoints</i>	
<code>prewarm($\mathcal{M}$) $\rightarrow \sigma$</code>	Reserve scratch and discover native skips
<code>ingest(τ, id, p)</code>	Validate and deduplicate, then queue decoded payloads in synchronous RL or stage compressed payloads in asynchronous RL
<code>apply(p)</code>	Scatter into scratch, then call the native loader

Table B.2 | Failures and how the protocol handles them, with the figures that illustrate each case.

Failure	Handling	Figure
Rejected <code>prepare</code>	Before admission, ends the transition; after admission, triggers overwrite recovery	B.1
Payload of an obsolete attempt	Dropped at reception	B.2
Invalid payload	The attempt fails before the payload is applied	B.2
Duplicate payload	Ignored if its bytes match; otherwise the attempt fails	B.2
Missing payload	The arrival check fails, followed by an overwrite retry	B.4
Receiver killed mid-attempt	Retired by <code>stop_and_wait</code> and rejoins after a dense refit; the overwrite retry updates the others	B.5, A.14
Lost commit reply	After the deadline, <code>read_commit</code> decides the outcome	B.4
Coordinator failure	Takeover drains the attempt, calls <code>read_commit</code> , then retries or ends with a conflict	B.6
Persistent failure	Operator abort; a closed gate stays closed until a dense refit	B.6

`wait_for_requests` and then calls `open_apply`, so version- v requests finish before any payload is applied. In asynchronous RL, they retain validated compressed payloads in a separate host staging area. After the coordinator confirms that every expected payload has arrived, closes the gate, and calls `open_apply`, a decoder feeds the bounded queue in declared order while

the application drains it in that order.

Transports. Both transports follow Algorithm B.1 but carry payloads differently. On object storage, each payload is uploaded as a multipart object. Payloads and the manifest remain in storage until cleanup, which follows commit or a completed `stop_and_wait`. The relay tree streams payloads over ZeroMQ DEALER/ROUTER sockets.

Reception checks. Receivers learn the version and identifier of the active attempt at `prepare`. Before queueing, they check that each payload belongs to the active attempt, so they drop late payloads from obsolete attempts. Receivers then check each payload's compressed-body digest, tensor names, shapes,

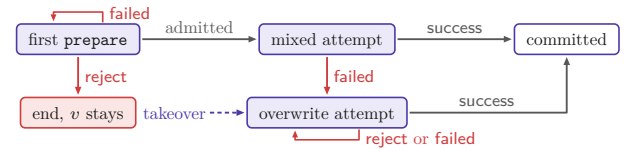


Figure B.1 | Admission and attempt modes. A rejected first `prepare` ends the transition with version v still committed, and a failed one is retried. After admission, any reject or failed exception leads to overwrite attempts, which repeat until commit, a conflict, or an operator abort. A coordinator that takes over an admitted transition starts in overwrite mode.

Algorithm B.1 Concurrent streaming schedule of one attempt. Failures propagate to Algorithm B.2.

Require: Prepared $\tau, g, v, m, W^{v+1}, B^v, \mathcal{M}$, bounded queues

Ensure: ($ack, success$) or ($\perp, failed$)

```

1: spawn owners
2: for all buckets  $\mathcal{E}_\ell$  streamed by build_complete do
3:    $(p_\ell, \sigma_\ell) \leftarrow \text{encode}(\mathcal{E}_\ell, m)$ 
4:   if  $\sigma_\ell \neq success$  then
5:     raise failed
6:   end if
7:   send( $\tau, p_\ell$ )
8: end for
9: spawn receiver ingestion
10: for all arriving ( $id, p$ ) do
11:   ingest( $\tau, id, p$ )
12: end for
13: spawn receiver application
14: wait until the coordinator calls open_apply( $\tau$ )
15: if asynchronous RL then
16:   spawn a decoder for staged payloads in declared order
17: end if
18: for all  $p$  yielded by the bounded queue in declared order do
19:   apply( $p$ )
20: end for
21: if asynchronous RL then
22:   await all sends
23:   wait for delivery and forwarding
24:   raise failed if any expected payload is missing
25: end if
26: wait_for_requests( $\tau, v$ )
27: open_apply( $\tau$ )
28: await all sends
29: return flush( $\tau$ )
    
```

dtypes, schema, and loader classification. If any of these checks fails, the attempt fails before that payload is applied. Receivers key each payload by its attempt, owner, and payload identifiers, apply it at most once, and ignore identical duplicates. Reusing an identifier with different bytes fails the attempt. Figure B.2 summarizes these checks and their outcomes.

Application order. After these reception checks, all receivers apply payloads in the single order that the payload identifiers declare for the attempt rather than in arrival order (Figure B.2). The staging queue reserves a slot for the next payload to apply, so later payloads cannot fill it first. Receivers may reorder only independent loader operations that have no cross-rank collectives or cross-payload side effects. Invalid payloads, conflicting canonical values, or conflicting destination writes stop application and fail the attempt, whether it is the first attempt or a retry.

Flush. The **flush** call closes the attempt to new sends, waits for delivery and relay-tree forwarding, and checks that every expected payload arrived: every payload in the manifest on object storage, or every

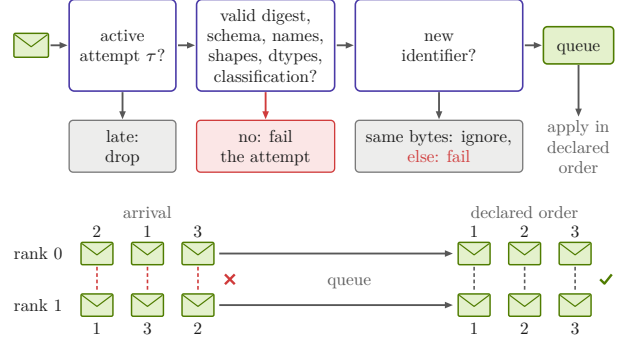


Figure B.2 | Top: checks on each received payload. An obsolete attempt’s payload is dropped, a failed check fails the attempt, and a duplicate payload is ignored when its bytes match and fails the attempt otherwise. Bottom: application order on two receiver ranks, with a loader collective per payload drawn as a dashed link. Payloads arrive in different orders, so applying them on arrival would pair different payloads in a collective. Applying them in the attempt’s declared order pairs the same payload on both ranks.

payload the owners sent on the relay tree. In asynchronous RL, the same arrival check also runs before the request gate closes, and **flush** repeats it afterward. Relay-tree queueing acknowledgments and the immediate responses to control calls confirm only queueing or registration, so **flush** also drains the receiver queues, synchronizes devices, and collects completion acknowledgments that report the configured post-apply checks. Figure B.3 contrasts the two kinds of acknowledgments. A failed **flush** returns ($\perp, failed$), which Algorithm B.2 treats as a failed exception.

B.2. Joint commit and recovery

Version v stays committed through every failure and retry until a joint commit publishes version $v+1$ after every required receiver acknowledges completion.

Commit and end of the transition. After **flush** succeeds with completion acknowledgments from every required receiver, **commit** may atomically replace K^v with $(v+1, g)$, provided the commit record still equals K^v . Owners start the in-place baseline update of § 6.3 when they read the commit record $(v+1, g)$, even when the commit is confirmed late or after failover. Each owner then reports completion to the coordinator, which ends the transition only after every owner has reported (Figure B.3). Receivers invalidate their KV caches before the coordinator reopens the request gate (Figure 8, bottom).

Commit outcomes. Commit outcome c is **success**, **failed**, **unknown**, or **conflict**. If the coordinator cannot

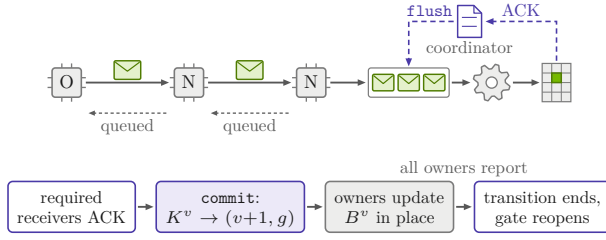


Figure B.3 | Top: acknowledgments on the relay tree of Figure 7. Each hop only acknowledges that it queued a payload. After **flush** drains the receiver queues and synchronizes devices, each receiver sends its completion acknowledgment, the ACK of Figure 4. Bottom: the end of a transition. After every required receiver acknowledges, **commit** replaces K^v with $(v+1, g)$. Owners read the new record, update their baselines in place, and report. Once all owners have reported, the transition ends and the request gate reopens.

confirm the outcome, **commit** returns **unknown**. Before the coordinator decides whether to retry after an **unknown** outcome or a failover, **read_commit**(τ) waits until no commit of τ can change the commit record. Each commit carries a deadline set by its coordinator. The control plane accepts it only before then and never after its attempt is marked obsolete, which bounds this wait (Figure B.4, bottom). The commit record $(v+1, g)$ confirms success, unchanged K^v confirms failure, and any other value signals a commit conflict, meaning that another coordinator committed the transition first. A conflict that **read_commit** reveals after the coordinator's own commit can reach only a replaced coordinator. It stops its attempt, cleans up, and returns without retrying, leaving the gate to the coordinator that committed. Of these outcomes, only a confirmed failure takes the retry path in Figure B.4, top.

Stopping an attempt. Before recovery, the coordinator calls **mark_obsolete**, after which endpoints ignore the attempt's later payloads and **open_apply**, and the control plane refuses the attempt's **commit** (Figure B.5). Then **stop_and_wait** stops delivery and relay-tree forwarding, waits for pending transport work and receiver writes to finish, synchronizes devices, and collects acknowledgments from all surviving endpoints. A terminated endpoint is retired from the failed attempt only after its process and pending writes are confirmed unable to resume. Receiver membership changes take effect only between attempts, so a retired receiver is not a required receiver of the retry (Figure B.5, bottom). After **stop_and_wait** returns, **cleanup** releases the attempt's payloads and delta records, leaving B^v unchanged. A failure before a successful **prepare** leaves nothing to stop or clean up.

Algorithm B.2 Refit with overwrite recovery. Retries stop at commit success, a conflict, an operator abort, or a rejection before admission.

Require: W^{v+1} , B^v , K^v , fixed $\mathcal{M}$ and source membership, and τ_{old} on takeover

Ensure: $v+1$ satisfying (3), or reject / conflict

```

1:  $m \leftarrow mixed$ ;  $admitted \leftarrow false$ 
2: if taking over an admitted transition then
3:   mark_obsolete( $\tau_{old}$ ); stop_and_wait( $\tau_{old}$ )
4:    $K \leftarrow read\_commit(\tau_{old})$ 
5:   if  $K \neq K^v$  then
6:     await baseline update named by  $K$ 
7:     cleanup( $\tau_{old}$ ); reopen the gate under  $\tau_{old}$ 
8:     return conflict
9:   end if
10:  cleanup( $\tau_{old}$ )
11:   $admitted \leftarrow true$ ;  $m \leftarrow overwrite$ 
12: end if
13: loop
14:    $\tau \leftarrow \perp$ ;  $c \leftarrow failed$ 
15:   try
16:      $g \leftarrow$  new identifier;  $\tau \leftarrow prepare(v+1, \mathcal{M}, g, m)$ 
17:      $admitted \leftarrow true$ 
18:      $(ack, \sigma) \leftarrow$  concurrent schedule in Algorithm B.1
19:     if  $\sigma \neq success$  then
20:       raise failed
21:     end if
22:      $c \leftarrow commit(\tau)$ 
23:   catch reject
24:     if  $\neg admitted$  then return reject
25:   catch conflict: return conflict
26:   catch failed: pass
27:   end try
28:   if  $c = unknown$  then
29:      $K \leftarrow read\_commit(\tau)$ 
30:      $c \leftarrow \begin{cases} success & K = (v+1, g), \\ failed & K = K^v, \\ conflict & otherwise. \end{cases}$ 
31:   end if
32:   if  $c = success$  then
33:     await baseline update named by  $(v+1, g)$ 
34:     cleanup( $\tau$ ); reopen the gate under  $\tau$ ; return  $v+1$ 
35:   end if
36:   if  $\tau \neq \perp$  then
37:     mark_obsolete( $\tau$ ); stop_and_wait( $\tau$ )
38:     cleanup( $\tau$ )
39:     if  $c = conflict$  then
40:       return conflict
41:     end if
42:   end if
43:   if  $admitted$  then  $m \leftarrow overwrite$ 
44: end loop

```

Failover and conflicts. Commit conflicts arise only from coordinator failover (Figure B.6): the coordinator that committed first is either a new coordinator started after failover or the old coordinator whose pending commit took effect. On takeover, the new coordinator first stops and drains writes from the old attempt τ_{old} , then waits with **read_commit** until no commit of τ_{old} can change the record. If such a commit took effect, the new coordinator waits for the baseline update named by the record, cleans up, reopens the gate under τ_{old} , and returns **conflict**. If the record remains K^v , the new coordinator cleans up

and retries in overwrite mode from the unchanged K^v , repairing any partial writes before its own commit replaces K^v . Its **prepare** replaces τ_{old} , so endpoints then ignore whatever the old coordinator still sends. The control plane also raises a **conflict** exception for any later **prepare** or **commit** of the old coordinator, which then returns without retrying or touching the request gate. An old coordinator whose commit outcome was **unknown** learns of the conflict from **read_commit**, stops and cleans up its attempt, and also leaves the gate alone. Owners hold their candidate shards and their baselines in memory, so they can finish the in-place baseline update after a coordinator failover. Because the update writes absolute values derived from the fixed W^{v+1} , an owner that reads the commit record again can safely repeat it. Failover therefore needs no durable record of which owners have finished.

Joining receivers. A new or restarted receiver restores the committed version through a dense refit, runs **prewarm**, and joins at the next **prepare** if the commit record still names that version. Otherwise, it first restores the newer committed version with another dense refit before joining.

Operator intervention. Some persistent failures fall outside the automatic recovery of Algorithm B.2 and require operator intervention: conflicting canonical values or destination writes, a write that bypasses the dispatch hook, a receiver that never acknowledges, or an owner that loses its candidate shards or its baseline in a crash. The operator aborts the transition: the coordinator stops and drains the attempt and

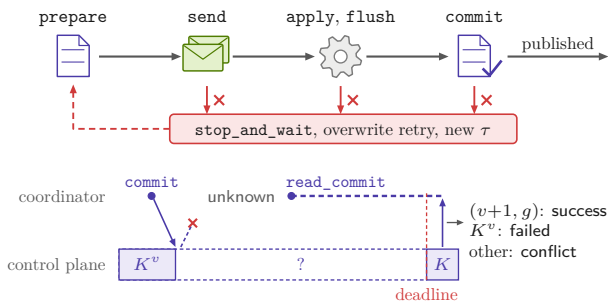


Figure B.4 | Top: the commit sequence. Version v stays committed until success, and a failure after **prepare** leads to **stop_and_wait** and an overwrite retry under a new τ . Bottom: resolving an unknown commit outcome, which takes the retry path only after **read_commit** confirms failure. The reply to **commit** is lost, so the record may hold K^v , $(v+1, g)$, or another value. **read_commit** waits until the deadline, after which the control plane accepts no commit of τ , so the record K it returns decides the outcome.

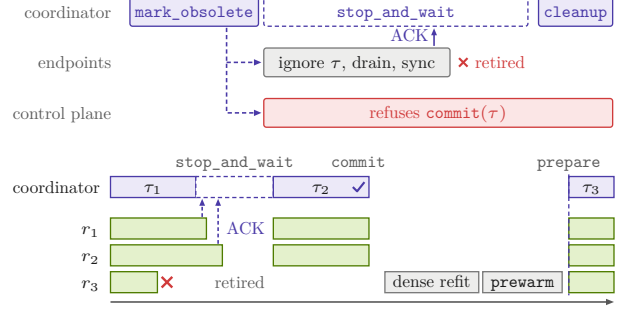


Figure B.5 | Top: the steps that stop a failed attempt τ . Once the coordinator calls **mark_obsolete**, the endpoints ignore the later payloads and the **open_apply** of τ , and the control plane refuses its **commit**. Then **stop_and_wait** waits until the surviving endpoints drain pending work, synchronize devices, and acknowledge, and it retires a terminated endpoint that cannot resume. **cleanup** then releases the attempt's payloads and delta records, leaving B^v unchanged. Bottom: receiver membership across attempts. Receiver r_3 fails during τ_1 and is retired, so it is not required in the retry τ_2 . After a dense refit and **prewarm**, r_3 joins the next attempt τ_3 at **prepare**.

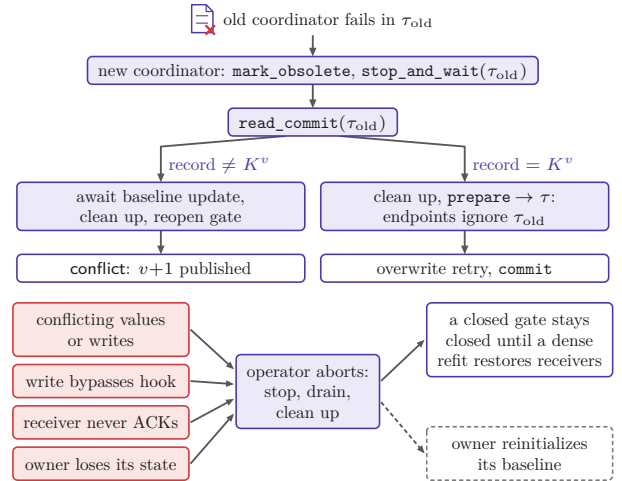


Figure B.6 | Top: coordinator takeover. After draining the old attempt, the new coordinator waits with **read_commit** until no commit of τ_{old} can change the record. It then retries in overwrite mode or ends the transition with a conflict. Bottom: operator intervention. Each persistent failure on the left makes the operator abort the transition, and a closed gate stays closed until a dense refit restores every required receiver to the committed version. An owner that lost its state also reinitializes its baseline, drawn dashed.

cleans up, and a closed request gate stays closed until a dense refit restores every required receiver to the committed version (Figure B.6). An owner that lost its state also reinitializes its baseline, as after a source membership change.

C. Integration and Measurement

This appendix describes the implementation behind § 3 and gives the pipeline settings for § 7.2 and measurement details for § 8. The cited NeMo RL commit [22] contains the Megatron Bridge and vLLM integration used in the experiments. Figure C.1 sketches the integration. Each transport adapter implements only `send`, `stop_and_wait`, and `cleanup` of Table B.1, and Appendix A.1 lists the training-side conversion tasks and mapping classes.

Rollout side. The rollout side adds three pieces to the native loader: a `prewarm` call, a dispatch hook, and a staging queue. For `prewarm`, receivers obtain canonical names, shapes, and dtypes from the training side. The dispatch hook intercepts the loader’s final storage copies, which are `aten.copy_` operations. The receiver reserves 32 resident groups of at most eight decoded payloads, plus one separate group of at most eight payloads under assembly, for at most 264 decoded payload slots in all (Figure C.2, bottom). In asynchronous RL, a separate host staging area holds compressed payloads until `open_apply`. A decoder then feeds this queue in declared order while the application drains it. Figure C.2 also shows the memory that a receiver keeps beyond its resident weights.

Pipeline settings. Owners compare source shards and convert required residual tensors in chunks, with targets of 64 MiB for object storage and 256 MiB for the relay tree, and both transports use a 512 MiB bucket target. The different chunk targets shift bucket boundaries, so the payload volumes of the two transports differ slightly in Table 4. These targets are measured in bytes of local source tensors for affine tasks and of converted canonical tensors for residual tasks, before unchanged elements are dropped. A bucket counts the full input size of each chunk that emits records. Tensors and chunks are never split, so a chunk or bucket can exceed its target. At 120B, the bucket target yields 482 payloads across 32 owners. Figure 8 illustrates both units on one owner, and Figure C.3 shows how their sizes count toward the targets.

Synthetic codec benchmark. The benchmark of § 8.2 runs PyTorch 2.14.0 on four CPU threads, with random seed 42 for the BF16 inputs and 44 for the FP32 noise. The same input and noise vectors serve all three values of a . Scaling the noise by a and adding it to the input are separate FP32 operations, and rounding precedes the bitwise comparison that selects changed words (Figure C.4). Each stream

holds only little-endian 16-bit words compressed with `zstd` 1.5.5 at level 1. The overwrite stream decodes to the updated BF16 words, and XORing the decoded masks into the inputs gives the same words. The reduction ratios of Table 2 use exact byte counts.

Table C.1 | Checkpoints of the latency runs.

Size	Checkpoint	GB
30B	Nemotron-3-Nano-30B-A3B	63.2
120B	Nemotron-3-Super-120B-A12B	247.2
235B	Qwen3-235B-A22B	470.2
550B	Nemotron-3-Ultra-550B-A55B	1,121
1T	Nemotron-3-Ultra-550B-A55B with twice the layers	2,242

Latency runs. Table C.1 lists the BF16 checkpoints behind the 30B–1T latency points, where the 30B point is Nemotron-3-Nano-30B-A3B, not the Qwen3-30B-A3B of §§ 8.3 and 8.4. Figure C.5 shows the testbed and the transport-only full-checkpoint reference of § 8.1.

Timed window and intervals. Each refit’s timed window runs from t_0 to t_1 . At t_0 , the first owner begins delta construction, and at t_1 , the last owner finishes its baseline update after the joint commit. Refit latency is $t_1 - t_0$ minus the post-apply check time. For the lower bounds of § 8.7, each compute interval runs from an operation’s start until its output is ready: change flags or locations for comparison, mapped indices and values for projection, canonical tensors for residual conversion, location and value buffers for encoding, and compressed bytes for `zstd`. For non-blocking operations, the interval ends when execution completes. Each relay-tree transport interval runs from payload submission until the sending endpoint, the owner’s transport endpoint, receives its queueing acknowledgment. On the relay tree, receivers stage and apply a payload only after their rollout node acknowledges its queueing, so no transport interval waits for its own payload’s staging or application. In the relay-tree latency runs, the staging queue never filled, so no interval waited for queue space either. Finalization runs from the end of the last loader application to t_1 . The `prepare` call precedes t_0 , and `cleanup` and the gate reopening follow t_1 .

Lower bounds. Each lower bound is the longest union of intervals on one owner or sending endpoint: the construction lower bound takes the union of each owner’s compute intervals, and the transport lower bound takes the union of each sending endpoint’s transport intervals. Before taking unions, we map Nsight Systems trace timestamps onto a common

```

# Transport adapter: three calls
class ObjectStorage:
    def send(self, tau, p):
        put(key(tau, p.id), p)
        post_manifest(tau, p.id)
    def stop_and_wait(self, tau):
        stop(tau)
        return wait_until_done(tau)
    def cleanup(self, tau):
        delete(tau)

# Training side: conversion tasks
for T in conversion_tasks(model):
    track(T) # every task
    if affine(T, M):
        # direct projection
        project(T)
    else:
        # residual path
        residual(T)

# Mapping metadata gives shard
# axes and offsets.
# No model-specific refit code.

class RelayTree:
    def send(self, tau, p):
        root.push(tau, p)
    def stop_and_wait(self, tau):
        stop_forwarding(tau)
        return collect_acks(tau)
    def cleanup(self, tau):
        release(tau)

# Rollout side: native loader
prewarm(M) # scratch, skips
on payload p: # either transport
    ingest(tau, p.id, p)
def apply(p):
    for name, idx, vals in p.items:
        scratch[:] = q
        scratch[idx] = vals
        with dispatch_hook():
            loader(name, scratch)
# No model-specific placement code.
    
```

Figure C.1 | Integration sketch. The codec, receiver checks, commit, and recovery of Algorithms B.1 and B.2 form a shared core with three interfaces: a transport adapter with three calls, conversion tasks with mapping metadata on the training side, and the serving runtime’s native loader with a dispatch hook on its final storage copies.

time axis and clip each interval to $[t_0, t_1]$, discarding empty intersections. Each union lies within the refit’s timed window, so both bounds are at most $t_1 - t_0$. We compute bounds per refit and average them as § 8.1 averages refit latency. Mean refit latency, which already excludes the post-apply check time, exceeds both mean bounds at every traced size. Figure C.5 illustrates both bounds in

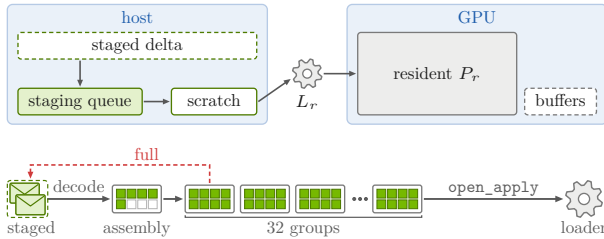


Figure C.2 | Top: receiver memory beyond the resident weights P_r . The host holds the decoded staging queue and the reusable scratch that **prewarm** sizes for the largest canonical tensor. Dashed boxes are temporary: the compressed delta staged in asynchronous RL, and GPU buffers for active masks and optional post-apply checks. Bottom: in synchronous RL, arriving payloads are decoded into a separate group under assembly, which then joins the queue of 32 resident groups of at most eight decoded payloads, the green cells. Thus the queue plus the assembly group holds at most 264 decoded payloads. After **open_apply**, the receiver applies queued payloads through the native loader. A full queue makes ingestion wait and slows upstream stages. In asynchronous RL, compressed payloads wait in the dashed host staging area until **open_apply**. A decoder then feeds the queue while the application drains it in declared order.

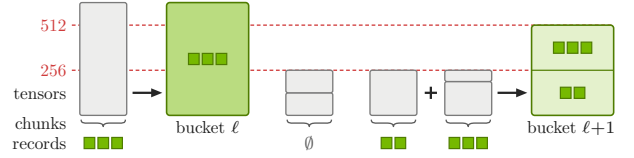


Figure C.3 | Chunk and bucket sizes, with heights proportional to input bytes and dashed lines at the relay-tree chunk target of 256 MiB and the bucket target of 512 MiB. A tensor larger than both targets forms its own chunk and bucket, since tensors and chunks are never split. Arrows lead from chunks with records to the bucket that counts their full input sizes, so the unchanged chunk ($\emptyset$) adds nothing to either bucket.

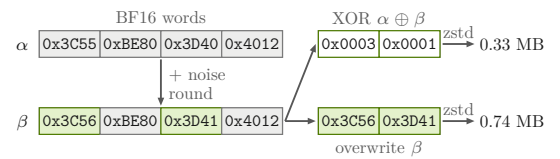


Figure C.4 | Synthetic codec benchmark. FP32 noise and rounding turn the BF16 inputs α into β , and only the changed words enter the XOR and overwrite streams. Sizes are those of Table 2 at $a = 10^{-4}$.

synchronous RL, and its receivers row shows the loader applications before finalization.

D. Coverage and Payload Format

For §§ 4, 5, and 7.1, this appendix details coverage, the payload format, and payload volume.

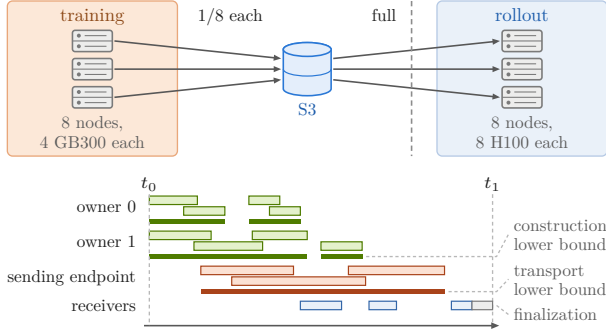


Figure C.5 | Top: latency testbed and transport-only full-checkpoint reference of § 8.1, with three of the eight nodes on each side drawn. Every training node uploads an eighth of the checkpoint to S3 concurrently, and every rollout node then downloads all of it in parallel across the dashed cluster boundary. Each node’s cross-cluster flow is up to 5 Gbps. Bottom: timed window of one refit. Thin boxes are intervals, and thick strips are their unions, which count overlaps once and exclude gaps. The longest union over owners gives the construction lower bound, and the longest union over sending endpoints gives the transport lower bound.

XOR and affine coverage. The XOR/overwrite split follows the mapping classes of Appendix A.1 and the loader paths of Appendix A.2. On the default BF16 paths, each owned shard of an affine tensor contributes one XOR item when it has changed elements and its path is representation-preserving with no overlapping writes. Each required residual tensor emits an overwrite item when its converted bits change. Figure D.1 summarizes the rule and shows why scaled and cast values need overwrites.

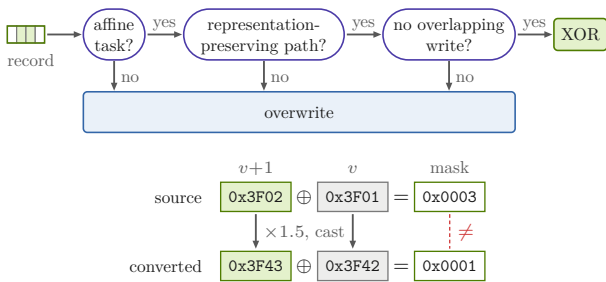


Figure D.1 | Top: encoding of a record in the default mixed mode. All conditions are decided in advance for each mapping class and loader path. Bottom: a source XOR mask under a conversion that scales by 1.5 and casts to BF16. The example element’s mask is 0x0003, but the mask of its converted values 0x3F42 and 0x3F43 is 0x0001, so the source mask cannot update the converted bits, which need an overwrite.

Direct-projection coverage. Table D.1 shows that affine mappings cover all but 185 canonical tensors of the Nemotron-3-Ultra-550B-A55B checkpoint,

which has no custom export hook. Routed experts use qualifying per-expert Auto mappings. Of the 185 residual tensors, 183 have mappings outside the supported classes: the Mamba input projections and convolutions, and attention Q/K/V, including those of the multi-token-prediction layer. The remaining two residual tensors are the embedding and output head. For Qwen3-30B-A3B, Table D.2 counts 18,721 of the 18,867 canonical tensors as affine. Its residual tensors are attention Q/K/V and the embedding and output head.

For the Qwen3-30B-A3B checkpoint of Table D.2, when every tensor and shard changes, XOR covers 18,721 of 18,867 items at tensor-parallel and expert-tensor-parallel degree one, 74,161 of 74,307 at degree four, and 148,081 of 148,227 at degree eight. Each of the 18,432 expert tensors and 48 attention output projections splits into t shards at degree t , while the 48 router and 193 normalization tensors stay whole, giving $18,480t + 241$ XOR items (Figure D.2). The XOR share of weight bytes stays 96.3% because a higher degree only splits the same weights into more shards.

For the Nemotron-3-Ultra-550B-A55B checkpoint of Table D.1, when every tensor and shard changes,

Table D.1 | Affine and residual canonical tensors in Nemotron-3-Ultra-550B-A55B. The Bytes column gives shares of the 1,121 GB checkpoint, with one share for the three residual rows together. Other affine weights include shared experts, routers, latent-MoE projections, attention output projections, normalization, and the output projections and state parameters of Mamba.

Weight type	Tensors	Bytes
<i>Affine</i>		
Routed expert up/down	50,176	93.86%
Other affine weights	662	3.13%
<i>Residual</i>		
Attention Q/K/V	39	3.01%
Mamba input projection and convolution	144	
Embedding and output head	2	

Table D.2 | Affine and residual canonical tensors in Qwen3-30B-A3B. The Bytes column gives each weight type’s share of the 61.06 GB checkpoint.

Weight type	Tensors	Bytes
<i>Affine</i>		
Expert gate/up/down	18,432	94.95%
Attention output projection	48	1.32%
Router	48	0.04%
Normalization	193	<0.01%
<i>Residual</i>		
Attention Q/K/V	144	1.65%
Embedding and output head	2	2.04%



Figure D.2 | XOR items of Qwen3-30B-A3B at tensor-parallel and expert-tensor-parallel degree t , drawn for $t = 4$. Each expert tensor and attention output projection splits into t shards, one item each, while each router and normalization tensor stays whole as one item.

403,869 of the 404,054 items are affine at tensor-parallel and expert-tensor-parallel degree eight. Each of the 50,176 routed expert up/down tensors splits into t_e shards at expert-tensor-parallel degree t_e , and 257 of the 662 other affine tensors split into t shards at tensor-parallel degree t , while the remaining 405 stay whole, giving $50,176t_e + 257t + 405$ affine items and 185 residual items. Every affine item takes the XOR path except those of the few Mamba state parameters that the native loader transforms, so the XOR share of weight bytes still rounds to the 97.0% affine share.

Payload format. Each payload carries attempt, owner, and payload identifiers, flat location and value buffers, and per-item metadata: tensor name, shape, dtype, encoding, location type, and offsets into both buffers. Contiguous locations use a start index, and the number of values gives the run length. Scattered locations use gaps in the smallest suitable integer type. A BLAKE2b-128 digest [1] covers each compressed payload body.

Figure D.3 illustrates the payload format with two tensors. The changed elements of tensor A form one contiguous run, so its metadata row stores a start index instead of location offsets and adds nothing to the location buffer. Tensor B’s scattered positions are stored as gaps. Both rows index the shared value buffer.

Payload volume. To relate payload bytes to the element change rate (Figure D.3), let Ω be the full canonical index set and $N = |\Omega|$. Because ownership is disjoint, $N_{\text{src}} = \sum_T |J_T|$ counts unique source elements and $N_{\text{chg}} = \sum_T |C_T|$ counts unique changes. The element change rate is $s_{\text{src}} = N_{\text{chg}}/N_{\text{src}}$. The emitted destinations form $D = \bigcup_{T \in \mathcal{T}_{\text{aff}}} \pi_T(C_T) \cup \bigcup_{u \in \mathcal{U}_{\text{req}}} D_u$. The destinations with changed canonical bits form $D_{\text{chg}} = \{d \in \Omega \mid I(W^{v+1})_d \neq I(W^v)_d\}$, and complete change detection makes it a subset of D . The canonical change density is $s = |D_{\text{chg}}|/N$. The element change rate and the canonical change density are equal when a representation-preserving bijection maps source elements to canonical elements.

Before zstd, the payloads of one attempt together carry metadata, run- and gap-coded locations, and

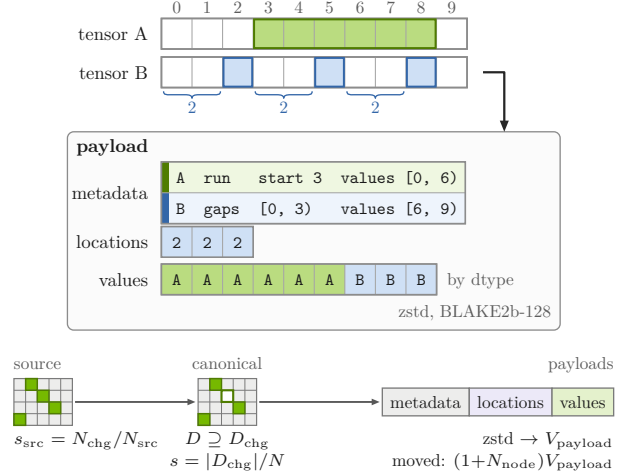


Figure D.3 | Top: payload format. To decode gaps, start at -1 and add each gap plus one: gaps 2, 2, 2 yield positions 2, 5, 8. Values have the width of their tensor’s dtype. Bottom: quantities behind payload volume. Owners find N_{chg} changes among N_{src} source elements, which gives the element change rate s_{src} . The emitted destinations D contain the changed destinations D_{chg} , whose share of the N canonical elements is the canonical change density s . When a mapping is not a representation-preserving bijection, D can also hold an unchanged destination, drawn outlined. Payloads carry metadata, locations, and values, and with N_{node} rollout nodes, object storage moves about $(1 + N_{\text{node}})V_{\text{payload}}$ per attempt in uploads and downloads.

one value per record. Let $H = \sum_{d \in \Omega} b_d$ be the canonical checkpoint size, where b_d is the width of element d in bytes. With one record per emitted destination, the value bytes total $\sum_{d \in D} b_d$, which at uniform width is $|D|H/N$, or sH when $D = D_{\text{chg}}$. Here D counts unique destinations. Each additional overwrite record for a destination adds another b_d bytes. More scattered changes need more location bytes, and more items need more metadata. After zstd compresses the locations and values, the attempt’s payloads total V_{payload} . On the object-storage transport, each attempt moves about $(1 + N_{\text{node}})V_{\text{payload}}$ of upload and download traffic when N_{node} rollout nodes download every payload. On the relay tree without retransmission, each payload crosses the cluster boundary once per attempt, so cross-cluster traffic is about V_{payload} , and the root forwards each payload over local links to every rollout node.

E. GRPO Experiment Settings

This appendix details the GRPO settings for the element-change measurements in § 2.2 and for the training runs in § 8.4.

Element-change measurements. These measurements follow the agentic SWE RL recipe for Qwen3 [24]: BF16 weights, FP32 AdamW state, constant learning rate 10^{-6} , and no warmup, weight decay, or KL penalty. The six measured models are Qwen3-30B-A3B, Llama-3.2-3B, Gemma-3-4B, and Qwen2.5 at 0.5B, 1.5B, and 7B.

Training runs. These runs use the first stage of the same recipe, with `swe1.jsonl` from the Nemotron-RL-Super-Training-Blends dataset. This stage rewards single-step tool calls for matching the arguments of the expert action. Each GRPO step uses 64 prompts with 8 responses each and a batch size of 512. With `ratio_clip_min=0.2` and `ratio_clip_max=0.28`, probability ratios are clipped to $[0.8, 1.28]$. The logged `train/gen_kl_error` estimates the per-token KL divergence between rollout and training policies. Figure E.1 shows the receiver kills.

F. System Comparison

This appendix gives the evidence behind each mark of Table 1 under requirements G1–G5 of § 2.3, and the same analysis for the slime framework.

NeMo-DCR. The design meets all five requirements. § 6.2 and Appendix A establish G1, and the dense-refit comparisons of § 8.1 confirm it. For G2, the payload volumes of Tables 3 and 4 scale with the changed set. For G3, placement stays in the native loader with no model-specific logic, as §§ 6.1 and 8.1 describe. For G4, both transports deliver without a cross-cluster collective, as § 7 describes, and the latency runs of § 8.1 use them between clusters that share no InfiniBand or EFA route. Payload deduplication, overwrite recovery, and joint commit meet G5, as § 6.3 and Appendix B describe, and with receivers killed mid-refit, both transports follow dense NCCL’s mean reward and KL trajectories in Figure 9.

Bit-exact (G1). Approaches A and B fully meet G1, and approach D avoids arithmetic reconstruction but meets G1 only partly. NCCL-Reshard

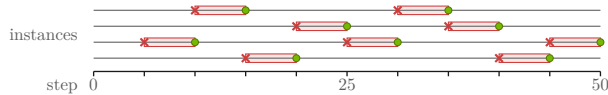


Figure E.1 | Receiver kills in one NeMo-DCR run of § 8.4, with an illustrative choice of instances. Every five steps, a randomly chosen vLLM instance is killed mid-refit at a cross and restarted five steps later, at a dot, where it restores the committed version.

and checkpoint-engine send every weight rather than a delta, so they preserve the target bits [20, 23]. TensorHub copies its published buffers into the serving runtime’s registered tensors, which yields exact target bits because the published buffers hold the weights in the layout and dtype of the registered tensors [40]. PULSE’s Appendix H.6 proves exact reconstruction of a tensor from a correct base [19], but extending the proof through a native loader requires conditions on the loader’s transformations and storage writes. In verl, NaN masks in intercepted copies protect unchanged positions, and a round-trip test checks that the model’s generated outputs are bit-identical [36], but the loader’s supported transformations are not stated. SparseRL-Sync patches the loader’s copy with a NaN-masked sparse scatter and reports tensor-by-tensor bitwise agreement with a dense refit, but does not specify supported transformations or interception of every storage write [11]. NeMo-DCR specifies both in the loader conditions of § 6.1 and Appendix A.2, and Appendix A proves that under these conditions its refits produce the same bits as a dense refit.

In approach C, arithmetic reconstruction of updated or pre-update values can introduce rounding errors. ROSE sends arithmetic differences [6]. AuroraRL reports lossless reconstruction by scatter-adding deltas but does not state when this reconstruction is bit-exact [30]. AReaL-DTE [26] reconstructs pre-update weights by inverting AdamW and compares BF16 bit patterns after aligning them to a common layout. Its authors note rounding in fused AdamW implementations, so comparison against these reconstructed pre-update weights can miss a changed element. NeMo-DCR instead compares against stored values retained in the committed source baseline.

Delta (G2). Approaches A and B move every weight byte, while the delta schemes in approaches C and D scale with the changed set. PULSE, SparseRL-Sync, and verl in approach D send absolute overwrites as indices and values of changed elements, with BF16 values in PULSE and verl. The verl framework detects changes on each training rank’s shard and maps them to full-tensor coordinates through block placements, and it supports conversions only when they permute elements [35, 36]. AReaL-DTE sends target values of detected changes, and AuroraRL packages each step as a BF16 delta checkpoint.

Native (G3). The serving runtime performs all weight placement in checkpoint-engine, SparseRL-Sync, and verl, and only part of it in NCCL-Reshard and TensorHub. In checkpoint-engine, the serving

runtime also chooses which weights to copy. NCCL-Reshard copies expert feed-forward shards directly into vLLM parameters using its own placement metadata and hooks. Only the remaining parameters pass through the native loader, so native placement is partial. TensorHub copies its published buffers into the serving runtime’s registered tensors without interpreting their layout. Yet weights are converted to the rollout layout before publication, not by the native loader, so native placement is only partial [40].

Most delta systems place updates themselves. ROSE uses shard-aware routing to receiver shards, and AuroraRL writes directly into parameter storage. PULSE writes values at the transmitted indices into receiver tensors that share the sender’s layout. AReaL-DTE remaps updates into rollout layouts at the receiver and uses the native loader only as a fallback, so its native placement is partial.

Decoupled (G4). NCCL-Reshard, SparseRL-Sync, and verl each use a collective that spans training and rollout. The checkpoint-engine system runs alongside serving runtimes, loads full weights onto GPUs from disk or the training engine, and broadcasts them among its workers. Its description mentions no cross-cluster transport [20]. PULSE, AReaL-DTE, ROSE, AuroraRL, and TensorHub transfer weights through replication, relays, or shared storage without a cross-cluster collective. TensorHub pipelines its full-weight replication, placing one replica in each remote datacenter before replicating locally. AuroraRL streams each delta checkpoint through one seed actor per region.

Recovery (G5). No existing system in Table 1 fully meets G5. ROSE, SparseRL-Sync, and verl describe no recovery protocol, although verl does checksum each broadcast payload. For NCCL-Reshard and checkpoint-engine, repeating a dense refit can restore weights, but the cited descriptions do not specify a complete protocol for partial failures and policy commit. TensorHub provides transactions, invalidates incomplete replicas, and recovers transfers from another source [40], but it coordinates version selection within each model-parallel group rather than committing across all required receivers. AReaL-DTE commits a version manifest and leaves partially applied versions to its control plane, without describing their repair. AuroraRL checks the version a delta applies to, uses leases for fault handling, stages deltas before activation, and lets lagging actors replay the delta chain, but does not describe repair after partial writes.

PULSE receivers advance independently, without committing the source baseline and the required receivers’ policy version together. PULSE’s Algorithm 5 checks

the local version before decoding [19], and a receiver applies consecutive deltas directly. PULSE also keeps periodic full checkpoints called anchors. After missed versions, the receiver restores a ready anchor and replays subsequent deltas in order. Ready markers prevent use of incomplete uploads, and a full-state hash checks each reconstruction before the new weights are returned. A mismatch triggers anchor recovery. Absolute overwrites can be repeated safely, and anchor restoration replaces a partially applied state.

slime. The slime framework’s delta weight sync publishes byte-level XOR or overwrite deltas of the gathered Hugging Face tensors to a shared filesystem [33]. Each rollout node patches a full local checkpoint, verifies per-tensor checksums, and reloads the checkpoint from disk through the native loader. This design meets G1–G4, but every refit reads the full model. It does not fully meet G5: the trainer advances its source baseline before any rollout node applies the published version, and no commit ties the baseline to the version that the rollout nodes hold.