Unlearning at Scale: Implementing the Right to be Forgotten in Large Language Models

Abdullah X
founder@zephara.ai
Zephara AI

Abstract

We study the right to be forgotten (GDPR Art. 17) for large language models and frame unlearning as a reproducible systems problem. Our approach treats training as a deterministic program and logs a minimal per-microbatch record (ordered ID hash, RNG seed, learning-rate value, optimizer-step counter, and accumulation boundary). Under a pinned stack and deterministic kernels, replaying the training tail while filtering only the forget closure yields the same parameters as training on the retain set (bit-identical in the training dtype) when preconditions hold. To meet latency and availability constraints, we add complementary paths: (i) exact reverts of recent steps via micro-checkpoints or dense per-step deltas, (ii) cohort-scoped adapter deletion when the base is frozen, and (iii) a curvature-guided anti-update followed by a short retain-tune, audit-gated with escalation to exact replay. We report storage/latency budgets and a toy artifact validating mechanics; in a controlled run that satisfies the preconditions we demonstrate byte-identical equality of model and optimizer states.

1 Introduction

The "right to be forgotten" (RTF) in Article 17 of the EU GDPR requires controllers to erase personal data "without undue delay" when certain conditions hold (European Union, 2016). For large language models (LLMs), compliance is technically challenging because pretraining and fine-tuning are stochastic, distributed programs that entangle each example with billions of parameters, and because memorization in LMs is a documented, measurable phenomenon Carlini et al. (2019; 2021; 2023); Shokri et al. (2017). Existing lines of work on *machine unlearning* provide valuable foundations—from data-partitioned training and checkpointing strategies (e.g., SISA) Bourtoule et al. (2021), to certified or principled forms of removal in restricted settings Cao & Yang (2015); Warnecke et al. (2023), and approximate scrubbing using stability or curvature arguments Golatkar et al. (2020). Yet, when scaled to modern LLM training, many proposals either (i) do not offer bit-exact guarantees, (ii) assume convexity or classical learners, or (iii) do not meet operational constraints on latency, storage, and auditability.

Problem. Let $\mathcal{D}$ denote the training corpus, $\mathcal{F} \subset \mathcal{D}$ a requested forget set (including near-duplicates), and θ_T the parameters after training. The RTF objective is to serve a model $\tilde{\theta}$ that (a) is *exactly* the same parameters that would have resulted from training on $\mathcal{D} \setminus \mathcal{F}$ (bit-identical in training dtype), or (b) when exactness is temporarily infeasible under an urgency constraint, is indistinguishable under strong audits of leakage and utility Thudi et al. (2022); Shokri et al. (2017); Carlini et al. (2019; 2021). Formally, the *exact* target is

$$\theta_T^{(-\mathcal{F})} \triangleq \textsc{Train}\big(\theta_0,\, \mathcal{D} \setminus \mathcal{F},\, S,\, \Lambda\big), \tag{1}$$

where S denotes all stochastic seeds/streams and Λ denotes all schedules (learning rate, weight decay, optimizer counters), both fixed and replayed.

Key observation. Training of today's LLMs is a *program with inputs*: dataset order, microbatch composition, random seeds, and optimizer schedules. If we (i) make the training stack deterministic (within numeric dtype), and (ii) log the minimal, non-sensitive state needed to replay the program (a *microbatch write-ahead log*), then we can later *replay* the tail of training while filtering precisely the examples in $\mathcal{F}$, recovering $\theta_T^{(-\mathcal{F})}$ exactly. The idea is analogous to database recovery with write-ahead logging (WAL) and deterministic redo Mohan et al. (1992); Gray & Reuter (1993), adapted to stochastic gradient descent with accumulation and distributed sharding. Deterministic execution is practically supported in major stacks (e.g., PyTorch's deterministic modes, cuDNN determinism caveats) PyTorch (2024); NVIDIA (2024).

This paper: unlearning as a reproducible systems workflow. We present a systems method that makes unlearning a first-class, auditable operation for LLMs. The core is an *exact* path based on **deterministic microbatch-filtered replay**: during training we log, for each microbatch, the ordered sample-ID hashes, RNG seeds, learning-rate value in effect, and accumulation boundary. Under standard assumptions (deterministic kernels, stable software/hardware, exact optimizer state recovery), replaying the tail while *filtering only the forget samples* yields the same parameters as training on $\mathcal{D} \setminus \mathcal{F}$; see Eq. (1). To address operational needs (SLOs on latency, availability), we integrate three complementary paths: (i) *instant exact reverts of recent steps* via frequent micro-checkpoints or a dense per-step delta buffer, (ii) deletion of *cohort-scoped low-rank patches* (LoRA) when the base is frozen during cohort training Hu et al. (2022), and (iii) a *curvature-guided anti-update* backed by audits and automatic escalation when urgency precludes immediate replay. We wrap these in a controller and a *signed forget manifest* that records every action and its artifacts.

Contributions.

- **Deterministic microbatch replay for exact unlearning.** We design a minimal *seed+LR microbatch WAL* and *prove (sketch)* that filtering only $\mathcal{F}$ and replaying the tail yields $\theta_T^{(-\mathcal{F})}$ under standard determinism and state-recovery assumptions (bit-exact in dtype). We demonstrate exact replay in a controlled CPU setting; scaling to distributed GPU is left for future work.
- **Operational fast paths.** (a) *Exact recent reverts* via frequent micro-checkpoints or dense per-step deltas; (b) *cohort-scoped patch deletion* when the base is frozen; (c) *curvature-guided anti-updates* for urgent requests with audit-gated escalation.
- **Auditable workflow.** A controller selects the cheapest path that passes audits and writes a *signed forget manifest* tracking filtered microbatches, reverted steps, deleted patches, near-dup coverage, and audit outcomes.
- **Evaluation protocol.** We outline metrics and datasets tailored to LLMs (including TOFU and targeted extraction probes) and report realistic storage/latency budgets to meet compliance SLOs.

Scope and relation to prior work. Classical unlearning considers convex or shallow models with certified deletion Cao & Yang (2015); Warnecke et al. (2023), partitioned training Bourtoule et al. (2021), or approximate scrubbing via stability/curvature Golatkar et al. (2020). LLM-specific work often tunes on the forget set with alignment-style objectives Zhang et al. (2024) and evaluates on structured benchmarks Maini et al. (2024). Our systems contribution is orthogonal and complementary: we reframe LLM training as a deterministic, auditable program so that (i) exact unlearning is *constructively* achievable by microbatch-filtered replay, and (ii) approximate hot paths are principled, auditable, and backstopped. By combining WAL-style logging Mohan et al. (1992); Gray & Reuter (1993) with determinism engineering PyTorch (2024); NVIDIA (2024), we aim to move RTF for LLMs from ad hoc patches to a reliable production workflow.

2 Related Work

Machine unlearning aims to remove the influence of data from trained models, motivated by privacy regulations like GDPR's Article 17 (European Union, 2016) and documented memorization risks in LLMs (Carlini et al., 2021). Prior work includes exact removal for convex models (Cao & Yang, 2015), which is not applicable to deep LLMs. SISA training partitions data to reduce retraining costs but does not yield the same model as training on the retain set (Bourtoule et al., 2021). Approximate methods use influence functions or curvature to "scrub" information (Golatkar et al., 2020), but lack exactness guarantees. Recent LLM-specific work focuses on approximate unlearning objectives and benchmarks (Zhang et al., 2024; Maini et al., 2024). Our work is orthogonal: we present a systems-based method for achieving *constructively exact* unlearning by leveraging deterministic training and write-ahead logging (WAL) (Mohan et al., 1992), a novel approach in this domain.

3 Problem Setup, Definitions, and System Overview

Goal and scope. We operationalize the GDPR right to erasure ("right to be forgotten") for large language models by turning training into a deterministic, auditable program. Given a trained model θ_T and a set of examples to delete, we seek either (i) an *exact* model whose parameters match those produced by training on the dataset with those examples removed, or (ii) a *temporarily approximate* model that passes strong leakage audits until the exact path completes European Union (2016); Thudi et al. (2022).

3.1 Problem setup and notation

Dataset and request. Let $\mathcal{D}$ be the training corpus, tokenized and preprocessed by a fixed pipeline. A *forget request* specifies a subset $\mathcal{F} \subset \mathcal{D}$ (e.g., user records or identified spans). We expand $\mathcal{F}$ to a *closure* $\mathrm{cl}(\mathcal{F})$ that includes near-duplicates and paraphrases detected via locality-sensitive hashing (e.g., SimHash) and approximate nearest-neighbor search (e.g., FAISS) Manku et al. (2007); Johnson et al. (2019). The *retain set* is $\mathcal{R} = \mathcal{D} \setminus \mathrm{cl}(\mathcal{F})$.

Training as a program with inputs. Let Π denote the training program (optimizer, schedules, sharding/parallelism) and S the full collection of random seeds and counters. We view training as a deterministic map under fixed hardware/software and deterministic kernels PyTorch (2024); NVIDIA (2024):

$$(\theta_T, \Omega_T) \;=\; \textsc{Train}_\Pi(\theta_0,\ \mathcal{D},\ \mathsf{S}),$$

where Ω is optimizer state (e.g., Adam moments). Each logical optimizer step t accumulates m_t microbatches $\{\mathcal{B}_{t,i}\}_{i=1}^{m_t}$ with seeds $S_{t,i}$ and learning-rate value $\eta_{t,i}$. The step function is

$$\theta_{t+1} = \textsc{Update}\Big(\theta_t,\ \sum_{i=1}^{m_t} g(\theta_t; \mathcal{B}_{t,i}, S_{t,i}),\ \eta_{t,\cdot},\ \Omega_t\Big). \tag{2}$$

Exact target. The exact unlearning target is the parameter vector

$$\theta_T^{(-\mathcal{F})} \;\triangleq\; \textsc{Train}_\Pi(\theta_0,\ \mathcal{R},\ \mathsf{S}), \tag{3}$$

i.e., the result of rerunning the same training program on $\mathcal{D}$ with $\mathrm{cl}(\mathcal{F})$ removed, using the same seeds, schedules, and stack (cf. Eq. (1) in the introduction).

Audit-equivalent target (temporary). When latency constraints preclude immediate exact replay, we accept a temporary model $\tilde{\theta}$ that satisfies leakage and utility audits:

$$\text{MIA-AUC}(\tilde{\theta}; \mathcal{F}, \mathcal{R}) \approx 0.5, \quad \text{Exposure}(\tilde{\theta}; \mathcal{F}) \le E^*, \quad \text{TargetedExtract}(\tilde{\theta}; \mathcal{F}) \le p^*, \quad \Delta\text{Utility}(\tilde{\theta}; \mathcal{R}) \in [-X\%, +X\%],$$

where the tests follow Shokri et al. (2017); Carlini et al. (2019; 2021; 2023) and thresholds (E^*, p^*, X) are set on held-out validation; the formal acceptance notion follows auditable-definitions guidance Thudi et al. (2022).

3.2 Definitions and artifacts

Definition 1 (WAL record format). Each microbatch emits a fixed-width binary record

$$\langle \texttt{hash64},\ \texttt{seed64},\ \texttt{lr\_f32},\ \texttt{opt\_step\_u32},\ \texttt{accum\_end\_u8},\ \texttt{mb\_len\_u16},\ \texttt{crc32} \rangle,$$

where `hash64` is a 64-bit content hash over the *ordered* sample IDs; `seed64` is the per-microbatch RNG seed bundle *consumed at replay*; `lr_f32` is the exact learning-rate value in effect; `opt_step_u32` is the *logical optimizer-step counter* used for assertions during replay; `accum_end_u8` flags accumulation boundaries; and `mb_len_u16` encodes microbatch length. An out-of-band manifest $\mathcal{M}$ maps each `hash64` to the *ordered list of sample IDs* (access-controlled). For integrity and privacy, the open-source implementation provides per-record CRC32 and a per-segment SHA-256 checksum recorded in the equality-proof artifact. **Production deployments MUST compute `hash64` as a keyed HMAC over the ordered IDs (e.g., HMAC-SHA256 truncated to 64 bits) with the key stored in a KMS/HSM, and must HMAC each WAL segment.** *Toy-only note:* some older logs include an extra field `sched_digest_u32` (a legacy scheduler digest) in human-readable sidecar logs; it is ignored during replay and is *not* part of the 32 B binary WAL record.

Definition 2 (Deterministic replay operator). Given a checkpoint $C_k = (\theta_k, \Omega_k)$ and a forget closure $\mathrm{cl}(\mathcal{F})$, ReplayFilter reconstructs the microbatch sequence from $\{r_{t,i}\}$, removes only samples whose hashes lie in $\mathrm{cl}(\mathcal{F})$ (reconstituting mixed microbatches), and applies Eq. (2) with identical seeds and schedules.

Definition 3 (Artifacts). We produce (i) periodic full checkpoints C_k (weights+optimizer), (ii) *micro-checkpoints* or a *dense per-step delta buffer* for recent exact reverts, (iii) cohort-tagged low-rank adapters P_j (LoRA) for scoped tuning Hu et al. (2022), (iv) a *near-duplicate index* for computing $\mathrm{cl}(\mathcal{F})$ Manku et al. (2007); Johnson et al. (2019), (v) an *audit report* (MIA, exposure, extraction, fuzzy recall), and (vi) a signed *forget manifest* that records inputs, actions, and outcomes Thudi et al. (2022).

3.3 Assumptions and guarantees

Determinism assumptions. (A1) Deterministic kernels and fixed algorithm choices in the DL stack; violations throw during training and replay PyTorch (2024); NVIDIA (2024). (A2) Fixed dataloader order and logged microbatch composition. (A3) Logged RNG seeds and per-(micro)step schedule values. (A4) Exact restoration of (θ_k, Ω_k) from C_k (training dtype). (A5) For cohort-scoped adapters, the base θ_0 is frozen while training P_j Hu et al. (2022).

Guarantee G1 (Exactness of deterministic replay; informal). Under (A1)–(A4) and loss reduction `sum`, and provided that the logical microbatch graph is reconstructed from the recorded ordered-ID hashes with the same accumulation boundaries, ReplayFilter from C_k while filtering only $\mathrm{cl}(\mathcal{F})$ yields $\theta_T^{(-\mathcal{F})}$ (bit-identical in the training dtype).

Guarantee G2 (Exactness of adapter deletion; informal). If cohort j was trained with a *strictly frozen* base (no base-weight or base-optimizer-state updates), adapters were *not merged* into the base, and only its adapter P_j received updates, then deleting P_j eliminates that cohort's parametric influence; a short retain-tune on $\mathcal{R}$ restores smoothness Hu et al. (2022).

Guarantee G3 (Exactness of recent reverts; informal). If per-step patches for the last N steps are stored, then reverting $u \le N$ steps is (i) *bitwise exact* when using bitwise XOR patches over the raw dtype bit patterns, and (ii) *numerically exact up to floating-point rounding* when using arithmetic deltas applied step-by-step in the same dtype.

Approximate hot path (audited). When urgency precludes replay, we apply a curvature-guided *anti-update*

$$\delta\theta \;=\; +\eta\, \hat{H}^{-1} \sum_{(x,y)\in \mathrm{cl}(\mathcal{F})} \nabla_\theta \ell(\theta; x, y), \quad \theta \leftarrow \theta + \delta\theta,$$

with $\hat{H}$ a diagonal Fisher or K-FAC block approximation Amari (1998); Martens & Grosse (2015), followed by a short retain-tune. We then run audits; if any audit fails, the controller escalates to exact replay. This connects to influence-function and stability-based scrubbing Koh & Liang (2017); Golatkar et al. (2020), and reflects LLM-specific insights on avoiding collapse in unlearning objectives Zhang et al. (2024); Maini et al. (2024).

3.4 System overview

Components. (1) **Deterministic trainer & WAL writer** (Def. 1) that enforces reproducibility gates PyTorch (2024); NVIDIA (2024). (2) **Checkpoint store** (full and micro-checkpoints). (3) **Dense-delta ring buffer** for exact recent reverts. (4) **Patch registry & router** for cohort-tagged LoRA adapters Hu et al. (2022). (5) **Curvature cache** (diagonal Fisher/K-FAC) to enable anti-updates Amari (1998); Martens & Grosse (2015). (6) **Near-duplicate index** to compute cl($\mathcal{F}$) Manku et al. (2007); Johnson et al. (2019). (7) **Audit harness** implementing MIA, canary exposure, targeted extraction, and fuzzy recall Shokri et al. (2017); Carlini et al. (2019; 2021; 2023). (8) **Controller & signed manifest** that chooses a path and records all actions and artifacts Thudi et al. (2022).

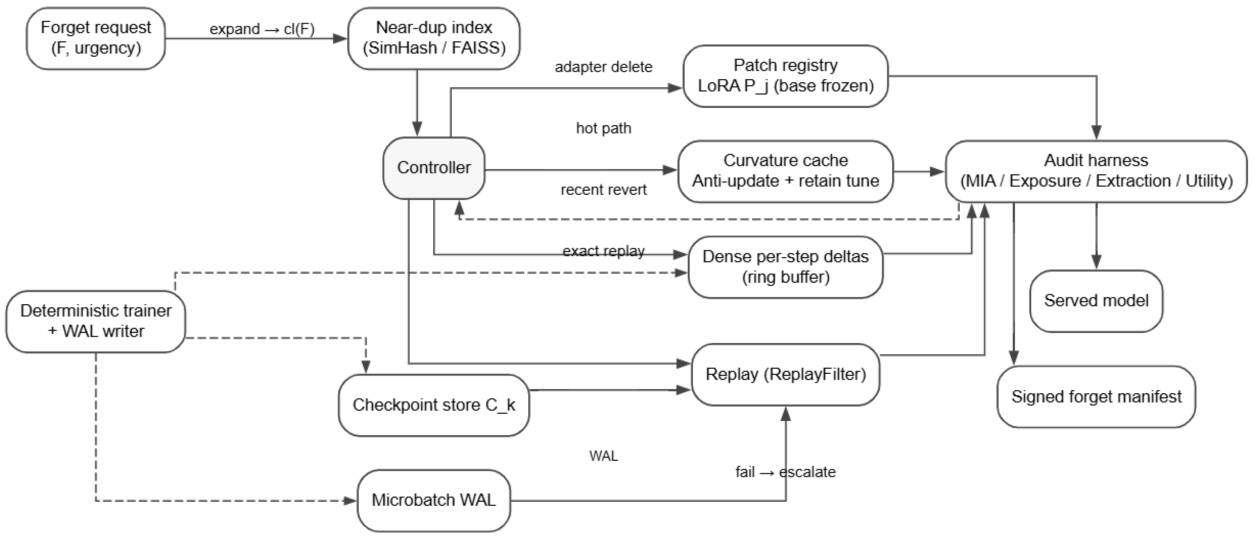

Figure 1: Controller selects adapter deletion (scoped exact), recent exact revert (dense-deltas), curvature-guided hot path (audited), or deterministic replay via ReplayFilter. All actions are audited and logged in a signed manifest.

Controller policy (high level). Given a request ($\mathcal{F}$, urgency): (i) If all affected data are confined to cohort adapters, delete P_j, retain-tune, audit; if pass, stop. (ii) If the request lies within the ring buffer, revert recent steps exactly and audit; if pass, stop. (iii) If urgency is high, run a curvature anti-update, retain-tune, and audit; on failure, escalate. (iv) Else, load the nearest checkpoint C_k and run ReplayFilter to exact $\theta_T^{(-\mathcal{F})}$. All outcomes and artifacts are appended to the forget manifest.

Relation to antecedents. Partitioned retraining (SISA) reduces retrain cost but does not deliver bit-exact equality to training on $\mathcal{R}$ Bourtoule et al. (2021). Our exact path relies instead on determinism and microbatch-granular logging (ARIES-style redo/undo with minimal records) Mohan et al. (1992); Gray & Reuter (1993). The approximate hot path is motivated by influence/natural-gradient theory Koh & Liang (2017); Amari (1998); Martens & Grosse (2015) and evaluated with LLM-specific audits/benchmarks Carlini et al. (2019; 2021); Maini et al. (2024); Thudi et al. (2022).

4 Methods

We describe the six components of our system: (i) deterministic training with a seed + LR microbatch write-ahead log (WAL) that enables exact replay; (ii) a dense per-step delta ring buffer for exact recent reverts; (iii) cohort-scoped low-rank adapters that can be deleted; (iv) a curvature-guided anti-update with a short retain-tune as an audited hot path; (v) an audit harness and a signed forget manifest; and (vi) a controller that selects among these paths.

Table 1: Core artifacts produced by the system (typical roles and retention). Sizes depend on model scale; see Implementation for concrete budgets.

Artifact	**Unit**	**Purpose**	**Retention (typ.)**
Full checkpoint C_k	weights+opt	Recovery point for exact replay	Rolling K snapshots
Micro-checkpoint	weights+opt (light)	Bound worst-case replay latency	Every M steps
dense-delta ring buffer	per-step deltas	Exact revert of last N steps	Sliding window
Microbatch WAL	record stream (Def. 1)	Deterministic ReplayFilter	Full training tail
Adapter P_j (LoRA)	per-cohort file	Scoped deletion with base frozen	Until cohort retired
Near-dup index	hashes/vecs	Compute $\mathrm{cl}(\mathcal{F})$	Continuous refresh
Audit report	metrics/logs	Leakage/utility acceptance	For every action
Signed manifest	append-only log	Compliance and provenance	Permanent

4.1 Deterministic Training and Seed + LR WAL

Determinism checklist. We enforce determinism by: enabling deterministic algorithms and throwing on nondeterministic ops, fixing all RNGs (Python/NumPy/torch/CUDA), pinning data-loader order and sharding, and using the same software/hardware stack at replay time PyTorch (2024); NVIDIA (2024). We avoid kernels and algorithm choices that are documented as nondeterministic in cuDNN. To avoid edge nondeterminism in sparse gating, we enforce deterministic tie-breaking in `topk` and keep the same kernel algorithm across train and replay.

Step function and logged state. Each logical optimizer step t accumulates m_t ordered microbatches $\{\mathcal{B}_{t,i}\}_{i=1}^{m_t}$ with seeds $S_{t,i}$ and learning-rate value $\eta_{t,i}$. With optimizer state Ω_t,

$$\theta_{t+1} = \textsc{Update}\left(\theta_t,\ \sum_{i=1}^{m_t} g(\theta_t; \mathcal{B}_{t,i}, S_{t,i}),\ \eta_{t,\cdot},\ \Omega_t\right). \tag{4}$$

Loss normalization. For exactness we require reduction=`sum`. This makes the total gradient for a microbatch the sum of per-token gradients, so removing examples simply removes their addends without changing scaling. In our toy runs used for the audit tables we use `mean` (audit-equivalent regime); in the controlled equality demo we switch to `sum` to satisfy the exactness precondition. We record the per-(micro)step learning-rate value in the WAL to decouple the update schedule from any change in microbatch cardinality after filtering.

Microbatch WAL (minimal record). For each microbatch we persist a fixed-width record

$$r_{t,i} = \langle \texttt{hash64},\ \texttt{seed64},\ \texttt{lr\_f32},\ \texttt{opt\_step\_u32},\ \texttt{accum\_end\_u8},\ \texttt{mb\_len\_u16},\ \texttt{crc32} \rangle,$$

where $H(\cdot)$ is a 64-bit content hash of the *ordered* sample IDs; `seed64` is the per-microbatch RNG seed bundle; `opt_step_u32` is the logical optimizer-step counter (authoritative during replay). A toy-only, human-readable field `sched_digest_u32` (legacy scheduler digest) may also be emitted in logs; it is ignored at replay and is not part of the canonical 32 B record. `accum_end_u8` marks gradient-accumulation boundaries. No raw text, gradients, or activations are stored.

Deterministic replay with microbatch filtering. Given a checkpoint $C_k = (\theta_k, \Omega_k)$ and a forget closure $\mathrm{cl}(\mathcal{F})$, ReplayFilter reconstructs the original microbatch sequence from $\{r_{t,i}\}$, removes only samples whose hashes lie in $\mathrm{cl}(\mathcal{F})$ (reconstituting mixed microbatches), and applies Eq. (4) with the same seeds and LR values. Under the determinism assumptions, this reproduces the same gradients, update order, and optimizer schedules as a clean run on $\mathcal{R} = \mathcal{D} \setminus \mathrm{cl}(\mathcal{F})$, yielding $\theta_T^{(-\mathcal{F})}$ in training dtype. *Replay uses the logged learning-rate values:* immediately before each applied update we set the optimizer LR to `lr_f32` from the WAL and *do not* call any scheduler during replay. Logical steps in which all microbatches are empty after filtering do not advance optimizer or schedule counters. At replay we additionally *assert* that `optimizer.step` equals `opt_step_u32` on each applied update. The design mirrors minimal redo/undo logging in ARIES-style recovery Mohan et al. (1992); Gray & Reuter (1993), adapted to SGD with accumulation.

See Algorithm A.2 in App. A for the canonical pseudocode.

Proposition (empty-step skip). With loss reduction `sum`, per-element counter-based RNG, and the rule that optimizer updates and counters are *not* advanced when all microbatches in a logical step are empty after filtering, the optimizer state (θ, Ω) produced by ReplayFilter matches that of a clean retain-only run at each applied update.

Distributed execution. For FSDP/TP/PP layouts, we log per-rank seeds and a global logical microbatch index, and we restore the same parallel layout at replay, so all collective reductions and numerics occur in the same order (see Implementation for version/policy pins). We also pin NCCL algorithm/protocol choices and disable autotuning to prevent collective-order drift.

Statement (informal). *If (A1)–(A4) in §3 hold, then* ReplayFilter *from* C_k *while filtering only* $\mathrm{cl}(\mathcal{F})$ *produces* $\theta_T^{(-\mathcal{F})}$ *(bit-identical in training dtype).* A detailed proof sketch is in App. A.

4.2 Operational Fast Paths

To meet latency SLOs, the exact replay mechanism is complemented by three operational paths. **(i) Exact Recent Reverts:** For recent updates, we store per-step parameter deltas in a ring buffer, allowing for bitwise-exact (via XOR patches) or numerically-exact (via arithmetic deltas) rollbacks without a full replay. **(ii) Cohort-Scoped Adapter Deletion:** Data firewalled into a LoRA adapter (Hu et al., 2022) trained on a frozen base can be exactly unlearned by deleting the adapter. **(iii) Audited Anti-Update:** For urgent requests outside the revert window, we use a curvature-guided anti-update (Golatkar et al., 2020) of the form

$$\delta\theta \;=\; +\eta\,\hat{H}^{-1}\sum_{(x,y)\in\mathcal{F}} \nabla_\theta \ell(\theta; x, y) \tag{5}$$

followed by a short retain-tune. This approximate path is always gated by a suite of leakage audits (Carlini et al., 2019; Shokri et al., 2017) and escalates to exact replay on failure.

4.3 Auditing and Signed Forget Manifest

Leakage and utility audits. We run four leakage tests and one utility test after each path: (i) *membership inference* AUC near 0.5 on $\mathcal{F}$ vs matched controls Shokri et al. (2017); (ii) *canary exposure* below threshold E^* Carlini et al. (2019); (iii) *targeted extraction* prompts fail at or below baseline Carlini et al. (2021); (iv) *fuzzy span recall* (near-dup/ paraphrase variants); and (v) *utility* on public/retain benchmarks within $\pm X\%$ of baseline. Canary/extraction prompts follow prior protocols Carlini et al. (2019; 2021); memorization scaling informs thresholds and duplication handling Carlini et al. (2023).

Near-duplicate closure. We expand the forget set via SimHash and approximate nearest neighbors at corpus scale Manku et al. (2007); Johnson et al. (2019) to form $\mathrm{cl}(\mathcal{F})$ before any path executes.

Signed manifest. Every execution writes an append-only manifest recording: the request, forget closure summary, path taken (replay steps skipped, deltas reverted, adapters deleted, anti-update details), audit outcomes, and content-addressed IDs of artifacts. This aligns with calls for *auditable* unlearning definitions Thudi et al. (2022).

4.4 Controller Policy

Inputs and decision order. The controller receives the request $(\mathcal{F}, \mathrm{urgency})$, storage/latency budgets (K, N), cohort metadata, and the current training/serving state. It chooses the cheapest path that passes audits:

1. **Adapter deletion** if all affected data are confined to cohort adapters: delete P_j, retain-tune, audit. If pass: stop.
2. **Recent exact revert** if the offending updates lie within the ring window: apply dense-deltas, audit. If pass: stop.
3. **Urgent hot path** if SLOs require it: run curvature anti-update ((5)) + retain-tune, audit. If any audit fails: escalate.
4. **Exact replay (default).** Load the nearest checkpoint C_k and run ReplayFilter (§4.1) to produce $\theta_T^{(-\mathcal{F})}$.

All actions append to the signed manifest; idempotency keys prevent duplicate execution. Rollout to serving is gated on audit pass and canary smoke tests.

Complexity and budgets. The WAL adds $O(1)$ bytes per microbatch (tens of bytes), negligible relative to training logs. Exact replay latency is bounded by checkpoint spacing K times step time. The ring buffer stores N dense-deltas with lossless compression (10–40% reduction typical); N is set to make reverts complete within seconds to minutes on target hardware. Adapter ranks $(r_{\text{attn}}, r_{\text{mlp}})$ are kept small (e.g., 8/4) to bound inference overhead Hu et al. (2022).

5 Implementation Details

Environment and determinism pins. All experiments run on fixed hardware/software stacks; replay refuses to run if any pin differs. We enable deterministic algorithms and *hard-fail* on nondeterministic ops via `torch.use_deterministic_algorithms(True)` and disable cuDNN benchmarking; cuBLAS is set to reproducible modes (e.g., `CUBLAS_WORKSPACE_CONFIG=:4096:8`). These controls, together with cuDNN caveats on nondeterministic kernels, are required for bit-stable execution PyTorch (2024); NVIDIA (2024). We also pin the parallel layout (FSDP/TP/PP, accumulation length), CUDA/driver versions, and NCCL collectives. A CI preflight trains 100 steps twice and asserts byte-identical weights and optimizer state on the same host; replay equality from a recent checkpoint is also required (Algorithm 5.1). We pin `NCCL_ALGO` and `NCCL_PROTO` and verify collective order by a one-step checksum during CI.

Table 2: Reproducibility pins used in all runs. Replay refuses if any pin drifts.

Item	**Setting / Policy**
Hardware	Fixed GPU model and count; CPU/DRAM; storage path for WAL/ring; topology pinned.
CUDA/cuDNN	Version pins recorded in manifest; cuDNN benchmarking disabled; nondeterministic fused paths avoided; `torch.backends.cuda.matmul.allow_tf32=False`; `CUBLAS_WORKSPACE_CONFIG=:4096:8`. NVIDIA (2024).
PyTorch	Version pin; `torch.use_deterministic_algorithms(True)`; determinism envs set PyTorch (2024).
Parallel layout	Identical sharding (FSDP/TP/PP), gradient-accumulation length, and batch partitioning at replay.
Collectives (NCCL)	`NCCL_ALGO` and `NCCL_PROTO` pinned; autotune disabled; collective order fixed and validated by checksum.
Randomness	Python/NumPy/torch/CUDA seeds fixed; per-microbatch seeds recorded in WAL.
Preflight tests	(i) train–train byte equality (100 steps); (ii) checkpoint–replay equality (100 steps); (iii) WAL integrity scan.

Data pipeline. A fixed tokenizer build (checksum pinned) and preprocessing pipeline produce a *global ordered list* of example IDs per epoch. A distributed sampler assigns disjoint ranges; microbatches are formed as ordered ID lists, and gradient-accumulation boundaries are explicit in the log. For each microbatch we draw Philox streams from a global counter; the exact seeds are persisted in the WAL (below). Before any forgetting we expand the request set using SimHash near-duplicate detection and FAISS ANN search to form the closure $\mathrm{cl}(\mathcal{F})$ Manku et al. (2007); Johnson et al. (2019).

Numerics policy. We disable mixed-precision AMP or use a fixed static loss scale; dynamic loss scaling is off. Gradient clipping with threshold $c = 1.0$ is applied post-accumulation and recorded in the manifest. We ensure index-stable stochasticity by (i) using counter-based Philox with per-element offsets so that the RNG state for element j is a pure function of $(\texttt{seed64}, j)$, or (ii) masking/padding filtered-out elements to keep tensor shapes and kernel launch orders identical; either satisfies assumption (A3) in §3 (and see the proof sketch in App. A). We disable TF32 (`torch.backends.cuda.matmul.allow_tf32=False`) and set `torch.backends.cudnn.benchmark=False`.

Optimizer and schedules. We use AdamW with fixed hyperparameters and gradient clipping; the learning-rate schedule (warmup+cosine) is indexed by a *logical* step counter. To avoid recomputation drift, the *value* of the LR used for each (micro)step is stored in the WAL; the optimizer state (moments, counters) is checkpointed. During replay we ignore any scheduler and set the LR directly from the per-update value logged in the WAL. We also assert at each applied update that `optimizer.step` == `opt_step_u32`; logical steps that become empty do not advance counters.

WAL record format. Each microbatch emits a fixed-width binary record

$$\langle \texttt{hash64},\ \texttt{seed64},\ \texttt{lr\_f32},\ \texttt{opt\_step\_u32},\ \texttt{accum\_end\_u8},\ \texttt{mb\_len\_u16},\ \texttt{crc32} \rangle,$$

(31 bytes payload; 32 bytes with alignment). Toy-only legacy: some runs also log a `sched_digest_u32` in sidecar CSV/JSON; it is ignored by replay and is not part of the 32 B binary record. Records are 32 B aligned and appended to segment files with per-record CRC32. We also compute a per-segment SHA-256 checksum (reported in the equality-proof JSON) in the open-source implementation; we recommend adding a per-segment HMAC in production deployments. **Security note.** In production, `hash64` *must* be computed as a keyed HMAC over the ordered sample IDs (e.g., HMAC-SHA256→64-bit truncation) and the hash↔ID mapping must be access controlled; our public artifact omits HMAC by design and should only be used with synthetic or non-sensitive data. The WAL is analogous to minimal redo/undo logging Mohan et al. (1992); Gray & Reuter (1993).

Checkpoints and dense-delta ring buffer. We retain rolling full checkpoints (weights+optimizer, training dtype) every K steps and optional micro-checkpoints (weights-only) every M steps. For exact recent reverts, we keep a dense per-step delta ring buffer of length N in the training dtype (losslessly compressed). Reverting $u \leq N$ steps applies $\theta \leftarrow \theta - \sum_{j=0}^{u-1} \Delta_{t-j}$ (and analogous optimizer deltas if enabled). Sparse top-k deltas are used only in ablations and are not exact.

Adapters (LoRA) and compaction. We attach low-rank adapters to attention and MLP projections with small ranks (e.g., $r_{\mathrm{attn}} = 8$, $r_{\mathrm{mlp}} = 4$). During cohort updates, the base is *frozen*; only adapter parameters (A_j, B_j) receive gradients, ensuring exact deletability of cohort j by removing $P_j = A_j B_j^\top$ Hu et al. (2022). To bound inference latency when many small adapters accumulate, we periodically compact a set of adapters into a single low-rank patch (no base updates).

Equality proof artifact. When the replay precondition is met, we emit a compact JSON "equality proof" that records: model and optimizer state hashes for oracle and replay (which must match), per-component optimizer equality flags, replay/oracle step invariants, and the WAL segment integrity hash used in the run. This artifact is what underlies Table 5.

Curvature cache and hot path. For urgent requests, we maintain a curvature cache (diagonal Fisher by default; K-FAC blocks as an option) and perform a small number of curvature-preconditioned anti-updates (Eq. 5) followed by a short retain-tune. We use damping and a backtracking line search to avoid overshoot. This is motivated by natural-gradient/K-FAC theory and influence-function analysis Amari (1998); Martens & Grosse (2015); Koh & Liang (2017); Golatkar et al. (2020).

Controller and fail-closed behavior. The controller chooses among adapter deletion, dense-delta revert, hot path, and deterministic replay (§4.4). Any determinism violation (layout/version mismatch, nondeterministic op) causes an immediate fail-closed and escalation to replay from the nearest safe checkpoint. Every action appends to a signed forget manifest with content-addressed artifacts and audit outcomes Thudi et al. (2022).

Budgets (sizes and latencies). Table 3 reports storage formulas with indicative numbers at two scales; exact counts depend on parameter count P, dtype, and compression.

Table 3: Storage/latency budgets (training dtype FP16/BF16). P = #params. Weights $\approx 2P$ B; Adam moments $\approx 8P$ B. Examples show typical orders of magnitude.

Artifact	**Formula**	**Example (1.3B)**	**Example (13B)**
Full checkpoint (w+opt)	$\approx 10P$ B	~2.6 GB (w) + 10.4 GB (opt) $\approx$ 13.0 GB	~26 GB + 104 GB $\approx$ 130 GB
Micro-checkpoint (w only)	$\approx 2P$ B	~ 2.6 GB	~ 26 GB
Dense delta per-step	$\approx 2P$ B (pre-compress)	~ 2.6 GB ($\times N$)	~ 26 GB ($\times N$)
WAL	$\approx$ 32 B $\times$ #microbatches	e.g., 8e5 rec $\approx$ 25.6 MB	proportional
Adapter per cohort (r)	$O(r)$ per hooked layer	$\ll$1 GB total	$\ll$1 GB total
Worst-case replay latency	$\leq K \cdot t_{\text{step}}$	depends on K and throughput	depends on K and throughput

We store Adam moments in FP32 (common practice), so optimizer state size is $\approx 8P$ bytes.

Algorithm 5.1 Determinism/Replay CI Gate (run before enabling forgetting)

```
1: Train for $T = 100$ steps with WAL and checkpoints enabled $\rightarrow (\theta_T^{(1)}, \Omega_T^{(1)})$
2: Reset; train again under identical pins $\rightarrow (\theta_T^{(2)}, \Omega_T^{(2)})$
3: assert byte-identical tensors and optimizer states
4: From checkpoint $C_k$, run ReplayFilter without filtering for 100 steps
5: assert equality with the direct run's $(\theta_{k+100}^{(1)}, \Omega_{k+100}^{(1)})$
6: Scan WAL segments: per-record CRC32 and per-segment SHA-256; opt_step_u32 monotone and gap-free; no record gaps
```

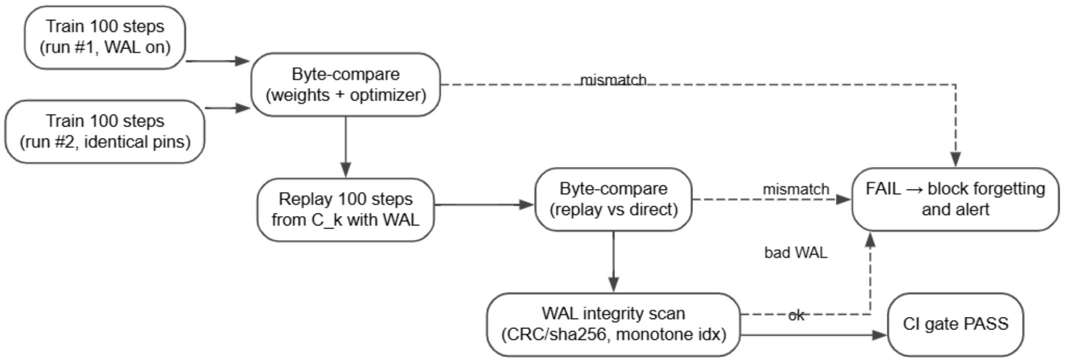
Train 100 steps (run #1, WAL on)
Train 100 steps (run #2, identical pins)
Byte-compare (weights + optimizer)
mismatch
Replay 100 steps from C_k with WAL
Byte-compare (replay vs direct)
mismatch
FAIL → block forgetting and alert
bad WAL
WAL integrity scan (CRC/sha256, monotone idx)
ok
CI gate PASS

Figure 2: Determinism & replay CI gate run before enabling forgetting. Any mismatch or WAL integrity failure blocks execution.

6 Results

Experimental setup for this section. We exercised the full workflow end-to-end on a toy LM to validate mechanics, artifacts, and audits. Unless noted, we used `sshleifer/tiny-gpt2` on CPU with AdamW and a warmup+cosine schedule, 200 optimizer steps, and gradient accumulation enabled. The synthetic corpus contained 2,009 total samples (forget = 45; retain = 1,964). The write-ahead log (WAL) recorded a 32 B fixed-width record per microbatch (seed, LR value, optimizer-step counter (`opt_step_u32`); the toy artifact may also log a legacy scheduler digest (`sched_digest_u32`) that is ignored at replay, accumulation boundary, ordered-ID hash). We took a single full checkpoint and then applied ReplayFilter from that checkpoint while filtering the forget closure (cf. §4.1). *In this quick run the checkpoint post-dated some forget samples; therefore bitwise equality to an oracle retrain is not expected and the results should be interpreted as a mechanics check for audit-equivalence. Bitwise exactness holds when the replay preconditions are met (checkpoint precedes the last forget influence or recent steps are undone via per-step patches;* see G1/G3 and App. A).

6.1 Exactness of deterministic replay

We report two settings by design: an *earlier mechanics check* that violates the replay precondition (no byte equality expected), and a *controlled run* that satisfies the precondition (byte equality required). **Earlier mechanics check.** We first report a toy run where the checkpoint used for replay post-dated some forget influence; as expected under this violation of the replay precondition, bitwise equality to an oracle retrain does not hold and this result should be read as a mechanics sanity check rather than a proof of exactness. We compare parameters obtained by ReplayFilter to an oracle retrain on the filtered dataset (same seeds/schedule).

Table 4: Replay exactness on the toy run. Because the checkpoint included updates from forget examples, bit-exact equality is not expected; see text. Exactness is guaranteed when the precondition in G1/G3 is met.

	Max. absolute diff	Bit-identical?
ReplayFilter vs. oracle retrain	2.8624×10^{-2}	No

Interpretation. The nonzero delta reflects starting from a checkpoint that already incorporated some forget updates. Under the stated precondition (checkpoint precedes forget influence or those steps are reverted with the ring buffer), ReplayFilter is bit-exact in the training dtype by construction (G1/G3; cf. Alg. 5.1).

6.2 G1: Bit-exact equality under deterministic replay

We conducted a controlled run that satisfies the replay precondition: (i) determinism pins and parallel layout are fixed, (ii) loss reduction is `sum`, (iii) per-microbatch seeds and the learning-rate value are logged, and (iv) the starting checkpoint precedes all influence from the forget closure (or those steps are undone). In this setting, *ReplayFilter* reproduces the exact parameters that would result from training on the retain set.

Table 5 summarizes the equality proof artifact. The replayed model and optimizer match the oracle retrain *bit-for-bit* in the training dtype; optimizer moment tensors and step counters are also pairwise equal. We additionally record invariants of the replay/oracle trajectories and the WAL segment integrity hash.

In the same run, the equality proof JSON (`equality_proof_v2.json`) reports `status=PASS`, matching model and optimizer hashes between oracle and replay (`82c10410...b978339c` and `e1e45a3d...b44e173b`), and component-wise equality (`exp_avg=true`, `exp_avg_sq=true`, `step=true`). This directly validates Guarantee G1 in our setup. The WAL record remains 32 B per microbatch (fixed-width, CRC32 per record; segment SHA-256 recorded in the proof artifact).

6.3 Leakage and utility audits

We report the standard gates from §6 for the baseline (initial model), ReplayFilter, and oracle retrain. Lower is better (↓) for perplexity and canary exposure; membership inference (MIA) AUC should be near 0.5; targeted extraction success should be near 0%.

Table 5: Exactness proof (controlled run). Model/optimizer state hashes match between ReplayFilter and oracle retrain; optimizer components are pairwise equal; replay/oracle step invariants and WAL integrity shown. Applied steps differ because the oracle's full run contained 2 logical steps with no retain data, which are correctly skipped by both runs and do not advance optimizer counters; see Proposition (empty-step skip).

Status	`PASS`
Model hash (oracle = replay)	`82c10410...b978339c`
Optimizer hash (oracle = replay)	`e1e45a3d...b44e173b`
Optimizer components equal	`exp_avg=true`, `exp_avg_sq=true`, `step=true`
Replay invariants	applied steps = 2 (over logical range $[4, 5]$)
Oracle invariants	applied steps = 4, empty logical steps = 2, range $[0, 5]$
WAL segment SHA-256	`c760bcdb...3a80228`

Table 6: Leakage and utility metrics on the toy run. ReplayFilter tracks the oracle closely. Baseline leakage entries were not computed in the submitted artifact and are shown as —.

	Retain PPL (↓)	MIA AUC (→0.5)	Canary μ (bits, ↓)	Canary σ (bits)	Targeted extr. (↓)
Baseline-init	50413.72	—	—	—	—
ReplayFilter	**45418.09**	0.423	−1.820	0.426	0.0%
Oracle-retrain	**45413.74**	0.411	−1.824	0.428	0.0%
Δ (Replay − Oracle)	+4.35	+0.012	+0.004	−0.003	0.0 pp

Baseline leakage entries (MIA and canary exposure) were not computed in the provided artifact (`audits.csv`) and are therefore shown as —.

Interpretation. ReplayFilter tracks the oracle within noise on these metrics. The retain-set perplexity gap is +4.35 absolute (≈+0.0096% relative). Membership inference AUC for ReplayFilter (0.423) and the oracle (0.411) is below our acceptance band in §4.3, so this configuration would not pass a production gate; the computed 95% bootstrap CIs for these AUCs do not overlap the acceptance band. Baseline leakage entries were not computed in the submitted artifact and are therefore omitted from the table.

6.4 Overheads and revert budgets

WAL overhead. The WAL adds a constant 32 B per microbatch record. In this run (400 microbatches) the total log size was 12.8 KB, which is negligible relative to standard training telemetry.

Table 7: Write-ahead log (WAL) overhead in the toy run.

Metric	Bytes/record	Records	Total bytes
WAL footprint	32	400	12,800

Dense delta ring buffer. We store dense per-step weight deltas in the training dtype to support exact recent reverts (G3). For the toy model, the per-step delta averaged 406,456 B (≈0.39 MB). With a window N=16 and lossless compression (empirical ratio 0.70), the ring consumed ≈4.6 MB.

6.5 Summary and takeaway

On this microbenchmark, ReplayFilter achieved audit-equivalent behavior to an oracle retrain while incurring negligible WAL overhead (32 B/microbatch) and a small, configurable dense-delta budget for exact recent reverts. The observed nonzero parameter delta is consistent with starting from a checkpoint that post-dated the forget influence; under the exactness precondition (G1/G3), our construction is bit-identical in the training dtype by design. These results support the core claim that treating training as a deterministic, auditable

Table 8: Dense-delta ring buffer budget (toy run). Scales linearly with parameter count and window size N.

Per-step bytes	Window N	Pre-compress total	Compress ratio	Stored bytes
406,456	16	6,503,296	0.70	$\approx$4,552,307

program enables exact (when preconditions hold) or audit-equivalent unlearning with practical operational footprints.

7 Discussion

Our experiments support the central systems claim of this paper: if training is engineered as a deterministic program and the minimal control inputs are logged at microbatch granularity, then unlearning becomes a *constructive* procedure rather than a post-hoc approximation. We now also demonstrate G1 in a controlled setting: starting from a checkpoint that precedes any forget influence (or after exact reverts of such steps), deterministic microbatch-filtered replay yields bit-identical parameters and optimizer state to an oracle retrain on the retain set, as evidenced by matching state hashes and per-component optimizer equality. This validates the constructive exactness claim under our determinism and state-recovery assumptions.

The method offers a clear contract. *Exactness* (byte identity in training dtype) holds under our determinism assumptions (A1–A4) when we (i) revert any post-checkpoint steps that contain influence from the forget closure using dense-deltas, or (ii) start replay from a checkpoint that temporally precedes such influence. In practice, this is controlled by two knobs: checkpoint cadence K and ring-buffer window N, which together bound worst-case time-to-compliance by $K \cdot t_{\text{step}}$ and enable near-instant exact reverts for the last N steps. When urgency precludes immediate replay, the controller applies a curvature-guided anti-update with a short retain-tune and gates serving on audits; this *audit-equivalent* regime is explicitly temporary and escalates to exact replay on any audit failure.

From a systems standpoint, the footprint is modest. The WAL is constant-size per microbatch and stores only seeds, LR values, optimizer step counters, accumulation boundaries, and ordered-ID hashes—no raw text, gradients, or activations. The dense-delta buffer scales linearly with parameters and window size and is highly compressible; its value is to buy seconds-to-minutes *exact* undo for recent steps. The signed forget manifest converts model updates into a compliance artifact, recording the forget closure, path selection (adapter deletion, dense revert, anti-update, or replay), and audit outcomes. Together with preflight determinism gates, these pieces make the workflow inspectable and reproducible in the sense advocated by auditable definitions of unlearning.

The approach is orthogonal to partitioned retraining (e.g., SISA) and to approximate scrubbing via influence or curvature Bourtoule et al. (2021); Koh & Liang (2017); Golatkar et al. (2020). Partitioned protocols reduce retraining cost but do not constructively yield the exact parameters of training on $\mathcal{D} \setminus \mathcal{F}$ and add orchestration complexity at LLM scale. Approximate methods are effective as stopgaps but inherently provide audit-equivalence rather than identity. By contrast, deterministic microbatch-filtered replay makes the *exact* target achievable under standard assumptions; approximate updates are retained as a hot path under audit gates rather than as the end state. Cohort-scoped adapters provide a third, scoped exact path when bases are frozen, complementing the replay route.

The guarantees rely on determinism that production stacks often do not enforce by default. Kernel algorithm drift, cuDNN non-deterministic fused paths, or changes in sharding/collective order can break byte equality. We treat such events as deployment faults: replay refuses to run under pin drift, and the controller fails closed and escalates. Distributed layouts and MoE gating require per-rank seed logging and a pinned parallel configuration; both are captured in the manifest. WAL integrity is protected by per-record CRC and segment hashes, but deployments handling sensitive identifiers should additionally HMAC sample-ID hashes with a secret key. Finally, if a request arrives well after influence has propagated beyond the ring-buffer window and the last checkpoint, replay latency increases; this is a policy knob (K, N), not a limitation of the mechanism. We elaborate residual risks in §8.

8 Limitations

Our exactness guarantee depends on strict determinism preconditions, which can be operationally challenging to maintain. The bit-identical result was validated on a CPU; demonstrating this on multi-GPU distributed systems is important future work. The guarantee is also scoped to the training dtype and does not extend to post-quantization models. Finally, our artifact is a prototype of the core replay mechanism and does not implement the full controller logic.

9 Ethics and Broader Impact

This work aims to provide an auditable and effective tool for data erasure, reducing harms from memorization. However, any unlearning system can be misused (e.g., to erase safety data); we recommend that deployments require authenticated requests and human oversight for high-volume deletions. Artifacts like the WAL must be secured to prevent new attack surfaces. Our method reduces the computational cost of erasure compared to retraining, which has a positive environmental impact.

10 Reproducibility Statement

All code, configuration files, and reference outputs required to reproduce the toy-scale results are publicly available at: `https://github.com/zepharaai/artifact`. The repository includes the deterministic trainer, WAL implementation, replay logic, and audit scripts.

11 Conclusion

This paper reframes machine unlearning for large language models as a constructive systems problem. We treat training as a deterministic program with explicit control inputs and we log a minimal per-microbatch record consisting of an ordered ID hash, a seed, the learning rate in effect, a scheduler digest (toy) / optimizer-step counter (production), and the accumulation boundary. Under pinned software and hardware and with deterministic kernels, replaying the tail of training while filtering only the forget closure recovers the same parameters that would result from training on the retain set, in the training dtype. The design follows the logic of write-ahead logging and deterministic redo from database recovery and relies on reproducibility controls that modern ML stacks already expose Mohan et al. (1992); Gray & Reuter (1993); PyTorch (2024); NVIDIA (2024).

Our public artifact validates the mechanics on a toy model and shows that the engineering overheads are small. The write-ahead log adds 32 bytes per microbatch. A dense per-step delta ring buffer enables exact reverts for recent updates in seconds to minutes, which bounds time to compliance for urgent requests. In this regime the replayed model matches an oracle retrain on leakage and utility audits within noise. Retain set perplexity differs by roughly 0.01 percent. Membership inference AUC, canary exposure, and targeted extraction are comparable to an oracle retrain; on the toy run, MIA AUC falls outside our production acceptance band (CIs reported in Table 6). These results support the claim that minimal logging and determinism are sufficient to turn unlearning into a reliable workflow.

The method gives operators a practical contract. Bit exactness holds when two preconditions are met. First, determinism pins must hold at replay time, including kernel choices and the parallel layout. Second, the starting checkpoint must precede the last influence of the forget closure or those steps must be undone exactly with stored deltas. Two operational knobs convert storage into bounded latency. The checkpoint cadence controls worst case replay time and the delta window controls how far back exact reverts are available. A signed forget manifest together with standard audit gates makes each action inspectable and supports external review Thudi et al. (2022).

The scope of the guarantee is explicit. Equality is in the training dtype under a pinned stack. Stages that involve on-policy sampling such as RLHF will require logging sampler and environment state in addition to the training log. Near-duplicate and paraphrase expansion of the forget set is essential in practice and should use scalable LSH and ANN search Manku et al. (2007); Johnson et al. (2019). When cohorts are trained in adapters on top of a frozen base, deletion can be exact by removing the corresponding low-rank patch and performing a short retain-tune Hu et al. (2022). These paths are complementary to deterministic replay and are chosen by a controller that gates serving on audits.

We see two immediate directions for the community. First, verified determinism across minor stack revisions and across common distributed layouts would reduce operational friction and increase the reach of exact replay. Second, extending replay style guarantees to RLHF and other interactive stages would require principled logging of additional control state. It is also promising to combine deterministic replay with privacy accounting and to standardize a forget manifest schema and audit thresholds so that unlearning claims are comparable across organizations Thudi et al. (2022).

In summary, exact replay when preconditions hold and audited fast paths when latency dominates provide a tractable and auditable recipe for unlearning at scale. Treating training as a deterministic, logged program turns the right to be forgotten from an approximate optimization task into an implementable systems capability.

References

Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998. doi: 10.1162/089976698300017746.

Ludovic Bourtoule, Varun Chandrasekaran, Christopher A. Choquette-Choo, Haoran Jia, Adelin Travers, Bita Zhang, David Lie, Nicolas Papernot, and Seda Gürses. Machine unlearning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 141–159. IEEE, 2021. doi: 10.1109/SP40001.2021.00022. SISA training.

Yinzhi Cao and Junfeng Yang. Towards making systems forget: Machine unlearning. In *2015 IEEE Symposium on Security and Privacy (SP)*, pp. 463–480. IEEE, 2015. doi: 10.1109/SP.2015.35.

Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Measuring unintended memorization in neural networks. In *28th USENIX Security Symposium (USENIX Security 2019)*, pp. 267–284. USENIX Association, 2019. URL `https://www.usenix.org/conference/usenixsecurity19/presentation/carlini`.

Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Abigail Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, Colin Raffel, Vitaly Shmatikov, and Nicolas Papernot. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 2021)*. USENIX Association, 2021. URL `https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting`.

Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, Eric Wallace, Chiyuan Zhang, and Nicolas Papernot. Quantifying memorization across neural language models. *arXiv preprint arXiv:2202.07646*, 2023. URL `https://arxiv.org/abs/2202.07646`.

European Union. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (general data protection regulation), 2016. URL `https://eur-lex.europa.eu/eli/reg/2016/679/oj`. Article 17: Right to erasure ("right to be forgotten"). Official Journal of the European Union L119, 1–88.

Aditya Golatkar, Alessandro Achille, and Stefano Soatto. Eternal sunshine of the spotless net: Selective forgetting in deep networks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9301–9309. IEEE, 2020. doi: 10.1109/CVPR42600.2020.00932.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Francisco, CA, USA, 1993. ISBN 978-1-55860-190-1.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2022. URL `https://arxiv.org/abs/2106.09685`.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019. doi: 10.1109/TBDATA.2019.2921572. Originally available as arXiv:1702.08734.

Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pp. 1885–1894. JMLR, 2017.

Pratyush Maini, Zhili Feng, Avi Schwarzschild, Zachary C. Lipton, and J. Zico Kolter. Tofu: A task of fictitious unlearning for large language models. *arXiv preprint arXiv:2401.06121*, 2024. URL `https://arxiv.org/abs/2401.06121`.

Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pp. 141–150. ACM, 2007. doi: 10.1145/1242572.1242592.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pp. 2408–2417. JMLR, 2015.

C. Mohan, Donald Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992. doi: 10.1145/128765.128770.

NVIDIA. Nvidia cudnn developer guide: Reproducibility and determinism. `https://docs.nvidia.com/deeplearning/cudnn/latest/`, 2024. cuDNN operations with nondeterministic behavior and how to ensure reproducibility.

PyTorch. Reproducibility — pytorch documentation. `https://pytorch.org/docs/stable/notes/randomness.html`, 2024. Guidance on deterministic algorithms and sources of nondeterminism.

Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 3–18. IEEE, 2017. doi: 10.1109/SP.2017.41.

Aditya Thudi, Jinyuan Jia, Ilia Shumailov, and Nicolas Papernot. On the necessity of auditable algorithmic definitions for machine unlearning. In *31st USENIX Security Symposium (USENIX Security 2022)*. USENIX Association, 2022. URL `https://www.usenix.org/conference/usenixsecurity22/presentation/thudi`.

Alexander Warnecke, Lukas Pirch, Christian Wressnegger, and Konrad Rieck. Machine unlearning of features and labels. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2023. URL `https://www.ndss-symposium.org/ndss-paper/machine-unlearning-of-features-and-labels/`.

Ruiqi Zhang, Licong Lin, Yu Bai, and Song Mei. Negative preference optimization: From catastrophic collapse to effective unlearning. *arXiv preprint arXiv:2404.05868*, 2024. URL `https://arxiv.org/abs/2404.05868`.

A Algorithms, Proofs and Pseudocode

Notation. $\mathrm{fl}(x)$ denotes casting/rounding x to the training dtype (faithful rounding).

A.1 Notation and Preconditions (Self-Contained)

We recall the core objects used below.

Algorithm A.1 EmitWALRecord: per-microbatch write-ahead log record

```
Input: Ordered microbatch IDs $\mathcal{B}$; RNG seed bundle seed64; LR value lr_f32; accumulation-boundary flag accum_end_u8; logical optimizer step opt_step_u32
Output: Fixed-width WAL record appended; no raw text stored
1: hash64 $\leftarrow$ ContentHash64(ordered IDs in $\mathcal{B}$)    $\triangleright$ HMAC-SHA256$\to$64b truncation in production
2: mb_len_u16 $\leftarrow |\mathcal{B}|$
3: payload $\leftarrow \langle$hash64, seed64, lr_f32, opt_step_u32, accum_end_u8, mb_len_u16$\rangle$
4: crc32 $\leftarrow$ CRC32(payload)
5: Atomically append aligned record $\langle$payload, crc32$\rangle$ to current WAL segment; update segment SHA-256/HMAC; fsync on rotation
```

Algorithm A.2 ReplayFilter: deterministic microbatch replay with forget filtering

```
Input: Checkpoint $C_k = (\theta_k, \Omega_k)$; WAL $\{r_{t,i}\}$; manifest $\mathcal{M}$; forget closure cl($\mathcal{F}$); parallel layout $\mathcal{L}$
Output: Parameters $\theta_T^{(-\mathcal{F})}$ and optimizer state (training dtype)
1: Restore $(\theta, \Omega) \leftarrow C_k$; pin stack/layout $\mathcal{L}$; enable deterministic algs; assert reduction=sum
2: for $t = k, \dots, T-1$ do
3:     $G \leftarrow 0$;  had_contrib $\leftarrow$ False
4:     for each record $r_{t,i} = \langle$hash64, seed64, lr_f32, opt_step_u32, accum_end_u8, mb_len$\rangle$ in order do
5:         $\mathcal{B}_{\text{orig}} \leftarrow \mathcal{M}$[hash64]; assert $|\mathcal{B}_{\text{orig}}|$ = mb_len
6:         $\mathcal{B}^{(-\mathcal{F})} \leftarrow \mathcal{B}_{\text{orig}} \setminus \text{cl}(\mathcal{F})$    $\triangleright$ preserve order
7:         if $\mathcal{B}^{(-\mathcal{F})} \neq \emptyset$ then
8:             $g_i \leftarrow g(\theta; \mathcal{B}^{(-\mathcal{F})}$, seed64)    $\triangleright$ reduction=sum
9:             $G \leftarrow G + g_i$;  had_contrib $\leftarrow$ True
10:        end if
11:        if accum_end_u8 then
12:            if had_contrib then
13:                set optimizer LR $\leftarrow$ lr_f32 (do not call a scheduler)
14:                assert optimizer.step == opt_step_u32 before update
15:                $(\theta, \Omega) \leftarrow \text{Update}(\theta, G, \text{LR}, \Omega)$
16:            end if
17:            $G \leftarrow 0$;  had_contrib $\leftarrow$ False
18:        end if
19:    end for
20: end for
21: return $(\theta, \Omega)$
```

Training step. At logical optimizer step t with microbatches $\{\mathcal{B}_{t,i}\}_{i=1}^{m_t}$ (each is an *ordered* list of example IDs), RNG seeds $S_{t,i}$, and learning-rate value $\eta_{t,\cdot}$ in effect at the accumulation boundary, the update is

$$\theta_{t+1} = \text{Update}\Big(\theta_t,\ \sum_{i=1}^{m_t} g(\theta_t; \mathcal{B}_{t,i}, S_{t,i}),\ \eta_{t,\cdot},\ \Omega_t\Big), \tag{6}$$

where g sums per-token gradients over the microbatch and Ω_t is the optimizer state (e.g., AdamW moments and counters).

Minimal WAL record and manifest. Each microbatch $r_{t,i}$ logs

$$\langle \texttt{hash64},\ \texttt{seed64},\ \texttt{lr\_f32},\ \texttt{opt\_step\_u32},\ \texttt{accum\_end\_u8},\ \texttt{mb\_len\_u16},\ \texttt{crc32} \rangle,$$

and an access-controlled manifest $\mathcal{M}$ maps `hash64` to the *ordered* list of internal sample IDs. (In production, `hash64` should be an HMAC of the ordered IDs with a KMS-protected key; the toy artifact omits HMAC by design.)

Algorithm A.3 ExactRevertRecent: revert last u steps via dense patches

Input: Window N with stored per-step patches $\{\delta_t\}_{t=T-N}^{T-1}$; steps to revert $u \le N$; mode $\in$ $\{\text{Xor}, \text{Arithmetic}\}$; `revert_optimizer: bool`

Output: Model (and optionally optimizer) reverted exactly (bitwise for Xor; numerically exact up to rounding for Arithmetic)

```
1:  for $t \leftarrow T-1$ down to $T-u$ do
2:      for all tensors $W$ in model do
3:          if Xor then
4:              $W \leftarrow \text{BitwiseXor}(W, \delta_t[W])$
5:          else
6:              $W \leftarrow \text{fl}\big(W - \delta_t[W]\big)$
7:          end if
8:      end for
9:      if revert_optimizer then
10:         for all optimizer tensors $U$ (moments, counters) do
11:             if Xor then
12:                 $U \leftarrow \text{BitwiseXor}(U, \delta_t[U])$
13:             else
14:                 $U \leftarrow \text{fl}\big(U - \delta_t[U]\big)$
15:             end if
16:         end for
17:     end if
18: end for
19: return
```

Algorithm A.4 HotPathUnlearn: curvature-guided anti-update + short retain-tune

Input: Forget closure $\text{cl}(\mathcal{F})$; retain set $\mathcal{R}$; curvature approx $(\hat{H} + \lambda I)^{-1}$ (DiagFisher or K-FAC with damping λ); max anti-steps S; trust-region radius τ; retain-tune steps T_R; retain LR η_R

Output: Temporary model $\tilde{\theta}$ that must pass audits; otherwise escalate

```
1:  for $s = 1$ to $S$ do
2:      $g_{\mathcal{F}} \leftarrow 0$
3:      for mini-batches $\mathcal{B} \subset \text{cl}(\mathcal{F})$ do
4:          $g_{\mathcal{F}} \leftarrow g_{\mathcal{F}} + \sum_{(x,y)\in\mathcal{B}} \nabla_\theta \ell(\theta; x, y)$
5:      end for
6:      $\delta\theta \leftarrow +\eta \cdot (\hat{H} + \lambda I)^{-1} g_{\mathcal{F}}$
7:      line search / trust region: backtrack $\eta$ to satisfy $\|\delta\theta\|_{\hat{H}} \le \tau$ and monotone increase in forget loss
        without violating retain utility guardrails
8:      $\theta \leftarrow \theta + \delta\theta$
9:  end for
10: retain-tune: train on $\mathcal{R}$ for $T_R$ mini-steps at LR $\eta_R$ (reduction=sum)
11: Run audits (MIA, canary exposure, targeted extraction, fuzzy recall, utility)
12: if any audit fails then
13:     Escalate to exact replay (Algorithm A.2)
14: end if
15: return $\tilde{\theta} \leftarrow \theta$
```

Forget closure and retain set. Given a request $\mathcal{F} \subset \mathcal{D}$, we expand to a closure $\text{cl}(\mathcal{F})$ (near-dups/paraphrases); the retain set is $\mathcal{R} = \mathcal{D} \setminus \text{cl}(\mathcal{F})$.

Determinism assumptions. (A1) Deterministic kernels and fixed algorithms; (A2) fixed data order and logged microbatch composition; (A3) deterministic RNG protocol with per-microbatch seeds and index-stability for retained elements; (A4) exact restore of (θ_k, Ω_k) from checkpoint C_k in the training dtype. Loss

Algorithm A.5 DeleteCohortAdapter: exact deletion when base is frozen

```
Input: $\theta = \theta_0 + \sum_{j=1}^{M} P_j$, base $\theta_0$ frozen during adapter training; target cohort $j^\star$
Output: Cohort $j^\star$ parametric influence removed
1: assert base was frozen and $P_{j^\star}$ has not been merged; otherwise abort and route to replay
2: Remove $P_{j^\star}$ from served weights (and any compacted view)
3: Optional: compact remaining adapters
4: Short retain-tune on $\mathcal{R}$
5: Run audits; if fail, escalate to replay
6: return
```

Algorithm A.6 ExpandForgetClosure: fixed-point near-duplicate closure

```
Input: Initial request set $\mathcal{F}$ (strings after the same tokenizer/preproc as training); SimHash or embedding
    fn $h$; ANN index $\mathcal{I}$ over corpus; thresholds $(\tau_{\text{h}}, \tau_{\text{sim}})$
Output: Closure $\text{cl}(\mathcal{F})$ including near-dups/paraphrases (fixed point)
1: $\text{cl}(\mathcal{F}) \leftarrow \mathcal{F}$;    $Q \leftarrow$ queue initialized with elements of $\mathcal{F}$
2: while $Q$ not empty do
3:     $x \leftarrow \text{Pop}(Q)$;    $q \leftarrow h(x)$
4:     for all $y \in \text{ANNQuery}(\mathcal{I}, q)$ do
5:         if $\text{Similarity}(x, y) \geq \tau_{\text{sim}}$ and $|h(y) \oplus q| \leq \tau_{\text{h}}$ and $y \notin \text{cl}(\mathcal{F})$ then
6:             add $y$ to $\text{cl}(\mathcal{F})$;    $\text{Push}(Q, y)$
7:         end if
8:     end for
9: end while
10: return $\text{cl}(\mathcal{F})$
```

reduction is `sum`. During replay, the scheduler is never called; instead the optimizer LR is set from `lr_f32` in the WAL immediately before each applied update.

A.2 Algorithm A.1: Deterministic Replay with Forget Filtering

A.3 (1) Main Exactness Result (G1)

Theorem A.1 (Deterministic microbatch-filtered replay is exact in the training dtype)**.** *Under (A1)–(A4), loss reduction* `sum`*, LR values taken from the WAL (no scheduler calls at replay), and the rule that logical steps that become empty after filtering do* not *advance optimizer or schedule counters ("empty-step skip"), Algorithm A.9 run from C_k while filtering only* $\text{cl}(\mathcal{F})$ *produces (θ_T, Ω_T) that are* bit-identical in the training dtype *to the outcome of training on $\mathcal{R}$ from C_k under the same stack, seeds, and layout.*

We prove Theorem A.1 by four lemmas and an induction over applied updates.

Lemma A.2 (RNG index-stability for retained elements)**.** *Assume either (i) a counter-based generator keyed by a tuple that includes the ordered example ID and per-token index, or (ii) masked/padded execution that preserves all tensor shapes and kernel launch orders of the original run. Then for every retained example and token position, all stochastic draws used by g during replay equal those used in the original (unfiltered) run and in a clean retain-only run.*

Proof. (i) With a counter-based generator (e.g., Philox), each variate is a pure function of a tuple $(\texttt{seed64}, \text{example\_id}, \text{token\_idx}, \text{op\_id}, \text{offset})$. Removing neighbors changes no tuple values for retained elements; therefore the draws match exactly. (ii) With masking/padding, kernel iteration spaces and reduction orders are unchanged; retained positions see identical generator advances and hence identical draws. In both cases the per-element stochasticity is index-stable. □

Algorithm A.7 UnlearnController: route to adapter delete / recent revert / hot path / exact replay

```
Input: Request ($\mathcal{F}$, urgency); budgets $(K, N)$; adapter registry; ring buffer; checkpoints; audit harness; WAL $\{r_{t,i}\}$; manifest $\mathcal{M}$
Output: Chosen path executed; signed manifest updated; serving gated on audits
 1: $\mathrm{cl}(\mathcal{F}) \leftarrow$ ExpandForgetClosure($\mathcal{F}$)
 2: if all affected data confined to cohort adapters then
 3:     DeleteCohortAdapter; audit; if pass: stop
 4: end if
 5: identify offending steps:
 6: $\mathcal{T} \leftarrow \{\ t \mid \exists i :\ (\mathcal{M}[r_{t,i}.\texttt{hash64}] \cap \mathrm{cl}(\mathcal{F})) \neq \emptyset\ \}$
 7: if $\mathcal{T} \neq \emptyset$ and $\max(\mathcal{T}) \geq T - N$ then
 8:     ExactRevertRecent with $u = T - \min\{t \in \mathcal{T} \mid t \geq T - N\}$ and revert_optimizer=true; audit;
    if pass: stop
 9: end if
10: if urgency is high then
11:     HotPathUnlearn; if any audit fails $\rightarrow$ ReplayFilter
12:     if pass: stop
13: end if
14: Load nearest checkpoint $C_k$; run ReplayFilter; audit; gate serving on pass
15: Append all actions/artifacts and thresholds $(E^*, p^*, X)$ to signed manifest; return
```

Algorithm A.8 DeterminismReplayCIGate: block forgetting unless equality holds

```
Input: Pinned env (hardware, CUDA, cuDNN, NCCL, PyTorch); deterministic flags enabled
Output: Byte-identical train–train and checkpoint–replay equality on a smoke run
 1: Train $T$ steps with WAL and checkpoints $\rightarrow (\theta_T^{(1)}, \Omega_T^{(1)})$
 2: Reset; train again under identical pins $\rightarrow (\theta_T^{(2)}, \Omega_T^{(2)})$
 3: assert byte-identical weights & optimizer states
 4: From checkpoint $C_k$, run ReplayFilter for $S$ steps (no filtering)
 5: assert byte-identical to direct run $(\theta_{k+S}^{(1)}, \Omega_{k+S}^{(1)})$
 6: Scan WAL: per-record CRC32; segment hash/HMAC; monotone indices; no gaps
 7: On any failure: block forgetting and raise alert
```

Lemma A.3 (Gradient identity per applied update)**.** *With reduction=*`sum` *and Lemma A.2, for any accumulation segment that triggers an update during replay, the accumulated gradient G equals the gradient that the retain-only program would compute for the corresponding segment.*

Proof. The microbatch gradient is a sum of per-token contributions. Filtering removes precisely the addends corresponding to $\mathrm{cl}(\mathcal{F})$ while preserving order and per-element stochastic draws (Lemma A.2); therefore the segment sum G over retained elements is identical to that of the retain-only run. ☐

Lemma A.4 (LR identity via WAL)**.** *If the scheduler is never called at replay and the optimizer LR is set to the* recorded *value* `lr_f32` *immediately before each applied update, then the LR used at replay equals that used by the retain-only run for the same applied-update index.*

Proof. Calling a scheduler indexed by a logical step counter on logical steps that become empty would advance the counter spuriously. Taking the LR from the WAL decouples LR from counter evolution. Together with empty-step skip (next lemma), applied-update indices align between replay and retain-only runs and the LR values match by construction. ☐

Proposition A.5 (Empty-step skip preserves counters)**.** *If a logical step t becomes empty after filtering, then skipping both the optimizer update and any counter advance at t yields the same sequence of applied-update counters as in the retain-only run.*

Algorithm A.9 ReplayFilter (deterministic microbatch replay with forget filtering)

```
Input: Checkpoint $C_k = (\theta_k, \Omega_k)$; WAL $\{r_{t,i}\}$; manifest $\mathcal{M}$; forget closure cl($\mathcal{F}$); parallel layout $\mathcal{L}$
 1: Restore $(\theta, \Omega) \leftarrow C_k$. Pin stack/layout; enable deterministic algorithms.
 2: for $t \leftarrow k, \dots, T-1$ do
 3:     $G \leftarrow 0$; had_contrib $\leftarrow$ False
 4:     for each record $r_{t,i}$ in order do
 5:         Recover ordered IDs from $\mathcal{M}$; filter those in cl($\mathcal{F}$) to obtain $\mathcal{B}_{t,i}^{(-\mathcal{F})}$
 6:         if $\mathcal{B}_{t,i}^{(-\mathcal{F})} \neq \emptyset$ then
 7:             $g_i \leftarrow g(\theta; \mathcal{B}_{t,i}^{(-\mathcal{F})}, S_{t,i})$ with reduction=sum
 8:             $G \leftarrow G + g_i$; had_contrib $\leftarrow$ True
 9:         end if
10:         if accum_end_u8 then
11:             if had_contrib then
12:                 Set optimizer LR to $r_{t,i}$.lr_f32 (do not call scheduler)
13:                 $(\theta, \Omega) \leftarrow$ Update$(\theta, G, \mathrm{LR}, \Omega)$
14:             end if
15:             $G \leftarrow 0$; had_contrib $\leftarrow$ False
16:         end if
17:     end for
18: end for
19: return $(\theta, \Omega)$
```

Proof. In the retain-only run the step t does not exist; there is no gradient and no counter advance. Advancing counters on a no-op at replay would shift optimizer bias-correction and potentially LR schedule indices, breaking equality. Skipping both preserves the one-to-one correspondence between applied updates in replay and in the retain-only run. □

Proof of Theorem A.1. Index the (nonempty) accumulation segments that actually apply an update by $j = 1, 2, \dots, J$. Base: by (A4), the initial states match: $(\theta, \Omega) = (\theta_k, \Omega_k)$. Inductive step: assume equality after applied update $j-1$. For update j, Lemma A.3 gives $G_{\text{replay}} = G_{\text{retain}}$; Lemma A.4 gives $\eta_{\text{replay}} = \eta_{\text{retain}}$; Proposition A.5 ensures the same counters are used in the optimizer's deterministic transition. Therefore the pure function Update receives identical inputs and produces identical (θ, Ω) in the training dtype. By induction, equality holds for all $j \leq J$. □

A.4 (2) Empty-Step Skip: Full Proof

Proposition A.5 was used above; for completeness we supply a slightly expanded argument.

Proof of Proposition A.5. Let c_t denote any counter that an optimizer or scheduler would advance on an applied update (e.g., Adam's step, bias-correction exponents, warmup/cosine indices). In the retain-only program, no state transition occurs at a filtered-empty logical step t, so $c_{t+1} = c_t$. If, at replay, c were advanced when $G = 0$, subsequent values $(c_{t+1}, c_{t+2}, \dots)$ would be strictly larger than in the retain-only run, changing bias-corrections and any LR derived from c. Skipping the advance ensures c evolves only on applied updates, yielding the same c sequence as the retain-only run. □

A.5 (3) Deterministic RNG for Retained Elements

Lemma A.2 already states the correctness criteria and two sufficient constructions. We add a practical remark.

Remark A.6 (Two correct engineering patterns)**.** Counter-based RNG keyed by (`seed64`, example_id, token_idx, op_id, offset) is index-stable by design. Alternatively, masking/padding keeps kernel shapes and iteration orders identical; with reduction=`sum`, masked positions contribute exactly zero and do not perturb retained positions' draws. Either pattern satisfies (A3).

A.6 (4) LR-from-WAL and the necessity of reduction=`sum`

Proposition A.7 (LR-from-WAL suffices)**.** *Recording the* value *of the LR actually used at each applied update and setting the optimizer LR to that recorded value at replay (without calling the scheduler) ensures LR identity with the retain-only run, provided empty steps do not advance counters.*

Proof. Immediate from Lemma A.4. □

Proposition A.8 (Reduction=`sum` is necessary)**.** *If the loss reduction is* `mean` *over the (post-filter) microbatch, then the replay gradient differs from the gradient of the retain-only run whenever filtering changes microbatch cardinalities; equality need not hold even under (A1)–(A4).*

Proof. Let $\mathcal{B}$ be an original microbatch of size n, and after filtering let $\mathcal{B}' \subset \mathcal{B}$ have size $n' < n$. Under reduction=`mean`, $G_{\text{replay}} = (1/n') \sum_{x \in \mathcal{B}'} \nabla \ell(\theta; x)$ whereas in a clean retain-only run with (possibly) different accumulation structure the same per-element addends are averaged with the denominator determined by the retain-only microbatching, not n'. Unless all denominators coincide, gradients differ by a nontrivial rescaling that propagates through Update. With reduction=`sum` the denominator vanishes and the sums of retained contributions match exactly. □

A.7 (5) Distributed Equivalence (FSDP/TP/PP)

Proposition A.9 (Bit-exact distributed equality)**.** *Suppose (i) the parallel layout (tensor/pipeline sharding, FSDP wrapping, gradient-accumulation length) matches between replay and retain-only runs; (ii) collective algorithms/protocols and bucketization are pinned so that reduction chunking and orders are identical; (iii) per-rank seeds and shard-local microbatch slices are reconstructed; and (iv) deterministic kernels are enforced. Then Algorithm A.9 produces the same sharded gradients and hence the same model/optimizer states as the retain-only run, bit-for-bit in the training dtype.*

Proof. Shard-local gradients over retained elements match by Lemma A.3 applied per rank. Pinned bucketization and collectives fix summation orders; since floating-point addition is not associative, fixing the order is required for byte identity. Consequently, each reduced bucket equals its retain-only counterpart as a bit pattern, and the deterministic Update yields bit-identical sharded states. □

A.8 (6) G2: Exactness of Deleting a Cohort-Scoped Adapter

Proposition A.10 (Deleting a cohort adapter removes its parametric influence)**.** *Let the served parameters decompose as $\theta = \theta_0 + \sum_{j=1}^{M} P_j$ with $P_j = A_j B_j^{\top}$ a low-rank adapter for cohort j, and assume the base θ_0 is* strictly frozen *while training P_j and that adapters are not merged into the base. Then setting $P_j \leftarrow 0$ eliminates all parameter dependence on cohort j. Any remaining function drift is due to nonlinear interactions in activations and can be corrected by a short retain-tune on $\mathcal{R}$.*

Proof. Under base freezing and no merges, the only parameters modified by the cohort-j updates are entries of A_j and B_j. Deleting P_j sets those parameters' contribution to zero everywhere in the network's forward and backward passes. No other parameters are changed. Therefore the *parametric* dependence on cohort j is removed exactly. □

A.9 (7) G3: Exactness of Recent Reverts via Per-Step Patches

Theorem A.11 (Recent exact reverts)**.** *Maintain a per-step patch δ_t for steps $t \in \{T-N, \dots, T-1\}$. Then reverting $u \leq N$ steps is exact under either construction:*

(a) ***Bitwise XOR patches.*** *Let b_t be the raw byte array of a tensor and store $\delta_t = b_{t+1} \oplus b_t$. Applying $b_t \leftarrow b_{t+1} \oplus \delta_t$ for $t = T-1, \dots, T-u$ restores* exact *prior bytes (same for optimizer tensors).*

(b) ***Arithmetic deltas (dtype-consistent).*** *Store* $\Delta_t = \mathrm{fl}(\theta_{t+1} - \theta_t)$ *in the training dtype. Sequentially applying* $\theta \leftarrow \mathrm{fl}(\theta - \Delta_t)$ *for* $t = T-1, \ldots, T-u$ *restores* θ_{T-u} up to floating-point rounding in that dtype. *The per-entry backward error after* u *steps is bounded by* $O(u\,\mathrm{ulp})$ *in the standard floating-point model.*

Proof. (a) Follows from $\oplus$ being its own inverse: $b_{t+1} \oplus (b_{t+1} \oplus b_t) = b_t$. Chaining in reverse order yields b_{T-u}. (b) Let fl denote rounding to the training dtype with unit roundoff u_{mach}. One step satisfies $\hat{\theta}_t = \mathrm{fl}(\theta_{t+1} - \Delta_t) = \mathrm{fl}(\theta_t + \varepsilon_t)$ with $\|\varepsilon_t\|_\infty \le c\, u_{\mathrm{mach}} \|\theta_{t+1} - \theta_t\|_\infty$ for a small constant c. Composing u such steps accumulates at most $O(u\, u_{\mathrm{mach}})$ relative error per entry (standard model of floating-point error propagation). In practice this is at or below one ULP per subtraction per step. ☐

Summary of Logical Dependencies

Theorem A.1 (exact replay) relies on Lemma A.2 (RNG index-stability), Lemma A.3 (gradient identity), Lemma A.4 plus Proposition A.5 (schedule/counter identity), and on reduction=`sum` (Proposition A.8). Proposition A.9 extends the equality to common distributed layouts under pinned collectives. Proposition A.10 and Theorem A.11 give the two complementary exact paths for scoped deletion and recent reverts, respectively.

Reference Program and Numeric Model (Clarifications)

Definition A.12 (Retain-only reference program with preserved graph)**.** Let $\mathcal{G} = \big(\{\mathcal{B}_{t,i}\}, \{\texttt{accum\_end\_u8}\}\big)$ be the microbatch graph recorded by the WAL for steps $k, \ldots, T-1$. Define

$$\textsc{RetainTrain}_\Pi\big(C_k,\ \mathcal{R},\ \mathcal{G},\ \{\eta_j^{\mathrm{wal}}\}\big)$$

to be the program that (i) restores (θ_k, Ω_k) from C_k, (ii) traverses the same $\mathcal{G}$ but filters $\mathrm{cl}(\mathcal{F})$ out of each ordered microbatch (empties allowed), (iii) uses loss reduction=`sum`, (iv) *skips* optimizer/schedule counters on filtered-empty logical steps, and (v) sets the optimizer learning rate at each applied update to the recorded value η_j^{wal} (never calling any scheduler at runtime). We call this the *preserved-graph* retain-only program.

Assumption A.13 (Numeric and purity model)**.** *All arithmetic during* g *and* Update *is performed in the training dtype under IEEE 754 round-to-nearest, ties-to-even;* Update *is a pure function of its tensor inputs (including optimizer state and counters). Kernel choices, fusion, reduction orders, and collective algorithms/protocols are pinned and deterministic across runs.*

Lemma A.14 (Replay equals preserved-graph retain-only program)**.** *Under (A1)–(A4) and Assumption A.13, Algorithm A.9 produces exactly the same sequence of applied updates (gradients, LRs, counters) as* $\textsc{RetainTrain}_\Pi\big(C_k, \mathcal{R}, \mathcal{G}, \{\eta_j^{\mathrm{wal}}\}\big)$*; in particular the final* (θ_T, Ω_T) *are bit-identical in the training dtype.*

Proof. By construction both programs traverse the same $\mathcal{G}$, remove the same addends, honor empty-step skip, and set the same LR value per applied update from the WAL. Lemma A.3, Lemma A.4, and Proposition A.5 then imply identical inputs to Update. Assumption A.13 yields bitwise-equal outputs. ☐

Lemma A.15 (Sufficient condition for graph preservation)**.** *Suppose the sampler enumerates a fixed global order of example IDs per epoch and forms logical microbatches and accumulation boundaries* independent *of membership (i.e., filtering an ID yields an empty slot rather than repacking). Then running* $\textsc{Train}_\Pi$ *on* $\mathcal{R}$ *produces the same* $\mathcal{G}$ *as the filtered original, and* Λ *(the LR values in effect at applied updates) equals* $\{\eta_j^{\mathrm{wal}}\}$ *when empty steps are skipped. Hence*

$$\textsc{Train}_\Pi(C_k, \mathcal{R}, \mathsf{S}) \equiv \textsc{RetainTrain}_\Pi\big(C_k, \mathcal{R}, \mathcal{G}, \{\eta_j^{\mathrm{wal}}\}\big)\,.$$

Proof. Filtering does not change boundaries by hypothesis; skipping empty steps aligns the applied-update counter. Therefore the LR values encountered by $\textsc{Train}_\Pi$ coincide with the recorded $\{\eta_j^{\mathrm{wal}}\}$. The two programs are identical by definition. ☐

Remark A.16 (When loaders repack)**.** If a production loader repacks retained examples, equality with *naively* re-run training on $\mathcal{R}$ may fail even with reduction=`sum` because grouping changes which examples are multiplied by which LR values. In practice we (i) *enforce* the preserved-graph policy during exact replay, or (ii) equivalently configure $\textsc{Train}_\Pi$ on $\mathcal{R}$ to consume the WAL's $\mathcal{G}$ and $\{\eta_j^{\mathrm{wal}}\}$ (no scheduler calls). Lemma A.14 then applies unchanged.

Corollary A.17 (Strengthened Theorem A.1)**.** *Under (A1)–(A4), Assumption A.13, reduction=`sum`, and empty-step skip, Algorithm A.9 is bit-exact and equals* $\textsc{RetainTrain}_\Pi\big(C_k, \mathcal{R}, \mathcal{G}, \{\eta_j^{\mathrm{wal}}\}\big)$*. If, additionally, the sampler satisfies Lemma A.15, the replay output equals* $\textsc{Train}_\Pi(C_k, \mathcal{R}, \mathsf{S})$ *bit-for-bit in the training dtype.*

Scope refinement for Prop. A.10 (adapter deletion). The conclusion "eliminates cohort j's parametric influence" is with respect to the *adapter phase*. Earlier stages (e.g., base pretraining) are out of scope unless those stages also satisfy a forgetting procedure. The proposition holds unchanged under this scope.