Replay Certificates for Cloud-Trained Branching Agents

From Lean-checked trace plans to independently replayable training updates

Anonymous review draft

Abstract

Cloud training can preserve a set of model weights while losing the evidence needed to reconstruct how branching agents produced their gradients. We propose a replay certificate that ties immutable episode events, typed latent-cache dependencies, policy action occurrences, and optimizer versions together. We prove that accepted prefixes retain their numerical identity through appends and rewinds, give a translation-validation criterion for equivalent forward and reverse plans, and bound likelihood-ratio error from per-action replay drift. We specify idempotent reverse contribution reduction and a commit protocol for exact-once optimizer publication under a linearizable head. A Lean 4 compiler and a separate replay auditor instantiate the reference design. On CPU the auditor reconstructs two steps of a small hybrid training run, verifies 16 seeded episodes, reproduces both optimizer checkpoint transitions, and detects altered scores and rewards. The design separates structural validity, numerical fidelity, seeded reproducibility, and authenticated provenance.

Keywords. Agent orchestration, replay certificates, translation validation, reinforcement learning, distributed recovery.

Contents

1 Introduction **3**
1.1 Claim levels . . . 3

2 Episodes as auditable causal objects **4**
2.1 Three identities . . . 4
2.2 State transition and local safety . . . 5
2.3 The two projections of a record . . . 5
2.4 Audit as a fiber of observations . . . 6

3 Compiler passes and translation witnesses **6**
3.1 Separating the controller from tensor terms . . . 6
3.2 Prefix stability . . . 7
3.3 Predecessor views and bridge insertion . . . 7
3.4 Type checking and register layout . . . 8
3.5 A translation-validation criterion . . . 9
3.6 Interpreting a fused kernel witness . . . 9

4 What a replay certificate can establish **10**
4.1 Integrity and provenance fields . . . 10
4.2 Structural and numerical audits . . . 11
4.3 A tolerance has an objective-level meaning . . . 11
4.4 Seeded regeneration . . . 12
4.5 Certificate checking order . . . 13

5 Reconstructable cuts and selective evidence **13**
5.1 Forward evidence across a cut . . . 13
5.2 Reverse evidence across a cut . . . 14
5.3 Membership commitments and coverage . . . 15
5.4 A lower bound on reconstructing discarded choices . . . 15

6 Recoverable updates across workers **16**
6.1 Typed contributions and deduplication . . . 16
6.2 An observable commit protocol . . . 17
6.3 Failure cases . . . 17
6.4 External effects and compensation . . . 18
6.5 Data-parallel training and placement . . . 19

7 The update ledger and reproducible reductions **19**
7.1 Sealing a group before replay . . . 19
7.2 Separating update completeness from numerical equivalence . . . 20
7.3 Finite-precision reduction as a witnessed choice . . . 21
7.4 What can be regenerated on demand . . . 21

8 Auditing a large collection of updates **22**
8.1 Measuring the audit and its blind spots . . . 22
8.2 Integrating a backend incrementally . . . 23

9 Reference implementation and evaluation **23**
9.1 Artifact layout and environment . . . 23
9.2 Audit procedure . . . 24
9.3 Observed results . . . 24
9.4 Mutation tests and claim boundaries . . . 25
9.5 Scaling the evaluation to a GPU cluster . . . 26

10 Conclusion **26**

A A complete audit witness **27**

B Running the reference checks **28**

C Implementation and proof boundary **28**

1 Introduction

Cloud training runs typically make a durable artifact from a temporary computation. The final model and optimizer state fit into a checkpoint. The active agents, their cached representations, their tool outputs, and the reasons particular action scores were included may disappear with the worker. A successful restart can establish that the worker resumed without discovering which cache edges or behavior probabilities had contributed to the last published update. A verification design should retain just enough information to reconstruct these dependencies and should attach a different claim to each level of evidence.

Octopus demonstrates the relevant difficulty [1]. One model coordinates forked agent threads through immutable radix cursors. A reader can borrow another branch's native key–value projections in selected attention layers, and a rewind can restore the reader's private recurrence while keeping its failed path in the training log. Prefix packing can evaluate a producer once while its value serves several readers. The same node can therefore have one policy owner, multiple numerical consumers, and no appearance in the final private context. A linearized chat transcript can hide that distinction. An optimizer checkpoint alone contains still less information about it.

This paper develops an executable audit design. It makes three new demands on a training artifact. First, the trainer should expose the transition log and enough provenance to reproduce the compiler's numerical and action plans. Second, it should distinguish graph acceptance from the ability to recompute the behavior probabilities and reproduce action draws. Third, it should publish checkpoints through a versioned protocol that gives a precise meaning to resuming or retrying an update. We prove the relevant properties of the design and run an auditor on an independently replayable CPU workload. The study uses the original report as a motivating system. Its evaluation concerns evidence and update integrity for a small hybrid model rather than the source's search accuracy.

The companion manuscript *Latent-Cache Programs for Branching Reinforcement Learning* develops the causal and smooth semantics of the underlying model. The present paper gives a separate interface for artifacts and replay. Its central results are the prefix stability and numerical translation criteria, a score-drift bound for audited behavior ratios, and a recovery invariant connecting complete reverse contributions to an observable optimizer version. The intended reader is an implementer deciding which checks can run at admission, at first replay, and after a crash.

1.1 Claim levels

A replay certificate is a finite set of witnesses plus a declared procedure for checking them. A witness can consist of raw data, a signed statement, a proof term, or a comparison of numerical outputs. The reference implementation uses trace records, immutable digests, a Lean compiler, and numerical comparisons. The word certificate here describes the data that support a stated predicate. Trust in a producer requires a verification key, a trust policy, and an authenticated binding to the relevant bytes.

A structural audit can establish that a log has no future read, that each append has a legal owner, and that rewinds name valid checkpoints. A numerical audit can establish that a compiled plan at the given behavior weights reproduces the recorded action scores within a stated norm and tolerance. A generative audit can use recorded randomness and a replayable environment to recreate choices and rewards. A provenance audit can authenticate a statement about where the

artifacts came from. Each is useful in its own setting. Their premises are independent enough that a useful audit records them individually.

Compiler testing, translation validation, and whole-compiler verification are related designs. Translation validation checks a produced translation for a particular input, while a verified compiler has an end-to-end theorem over its modeled source and target [2, 3]. We use a small Lean implementation and mechanical lemmas for integer layout and event-prefix replay. Our numerical argument also requires a backend contract, and the provided auditor exercises that contract for specific records. A bounded audit of a real-valued program is evidence about that record. It does not replace the general correctness proof or a stronger backend verification.

2 Episodes as auditable causal objects

2.1 Three identities

An execution record uses a unique tuple (r, v, g, e) to identify a run, behavior version, prompt group, and episode. The trace within the episode lists events in a topological order of controller dependencies. A numerical node has the ID of its append occurrence. Its data dependencies are its private predecessor and the producers of any visible borrowed slices. A score occurrence has identity (r, v, g, e, u, j) for node u and local token offset j when the token was sampled. Physical storage identifies a byte allocation in one cache domain. A successful implementation maps these three sorts into one another without identifying them.

In particular, a branch ID can survive a rewind while its private cursor changes. A tensor value can be kept across several reads while the reader's actions remain distinct. A mounted occurrence can be repeated in an attention list even though it shares one physical projection. These cases prevent using a content digest or a rotary index as a score ID. The digest identifies bytes under a chosen profile. The index identifies a numerical phase. The score ID identifies an event in the probability space.

Definition 2.1 (Sealed episode record)**.** A sealed record consists of an ordered event list with stable occurrence IDs, the branch and checkpoint names, the owned token deltas, the ordered borrowed slice requests, and the branch completion evidence. At each sampled token it contains an ordered legal support, a selected token in that support, and its behavior log-probability. The envelope supplies a behavior model digest, numerical contract, compiler profile, context or tool provenance, reward, and random draw profile. Fields that are unavailable are declared absent at their claim level.

The reference format records its small environment through a deterministic seed derivation and code digest. A networked retrieval environment should also record responses or durable references to them. For a fixed-input replay, a tool response is an external fact to which the audit assigns an identity. For a new causal intervention, a simulator or live environment must define a new response under changed inputs. Replaying the same response after changing the query is useful only under an explicit fixed-input estimand.

2.2 State transition and local safety

The host state includes a map from branch names to live cursor records, a map of published append nodes, and a chronological node strip. A cursor holds its private tail, its next native position, its ordered mount list, its current growth ledger, and a stack of restorable checkpoints. The compiler is a deterministic partial transition function over this state. An append records a node with these input ports and advances the private state. A fork gives a new branch an inherited cursor with a fresh budget. A read adds a foreign published range to the visibility list and adjusts the next position. A rewind restores an earlier stack entry. A wait requires the declared child completions. A close publishes a terminal mark. Unfulfilled waits can be scheduled earlier in a richer runtime, but they are emitted after their completion dependencies in the reference input language.

The separation between chronological and private order is an instance of causal partial ordering [4]. A read requires an earlier publication in chronological order. Rewind restores the private tail without putting any backwards edges into the numerical graph. To recover a reader's recurrence, store the exact private recurrent state and any convolution tail at the saved checkpoint. To recover its attention view, store the ordered list of borrowed occurrences and the associated native positions. These fields can differ between two views with the same tail. The branch's history ledger and action history are also separate. Decreasing the live growth count does not decrease the number of actions that were actually drawn.

For incremental training-graph construction, a policy may access its new mount only after a deterministic bridge token or another explicit pre-action computation under the new view. The reference compiler uses the bridge convention. A plan with a sampled first token after a visibility change must give a predecessor with the appropriate view. Merely finding an earlier logit array of the correct shape is insufficient. This is an example of a semantic invariant that a type checker alone cannot infer from tensor dimensions.

2.3 The two projections of a record

Define π_{data} to map a log to its ordered numerical graph and π_{action} to map it to the action ledger. Both are functions of one immutable source. They forget different information. The data projection may keep forced observations and borrowed slices, while the action ledger uses only sampled occurrences and points to the appropriate predecessor outputs. A trace can also project to terminal private contexts. That projection forgets discarded work and the placement of sampled stop or rewind decisions.

For a finite accepted trace, write $\mathcal{A}(\tau)$ for its sampled occurrences. A policy objective can select a particular weighting w_a for each occurrence. The stable identity keeps multiplicity separate from that weighting. A weight of one per occurrence corresponds to the trace likelihood score under parameter-independent external transition kernels. A normalized clipped objective can divide by $|\mathcal{A}(\tau)|$ while still assigning one owner to each sample. Dataflow fanout affects derivatives of each log-probability, and action ownership affects the count of the log-probability terms. These are separate axes of the audit.

Example 2.2 (Equal final histories, different updates)**.** Two trials start at the same cursor, take different sampled detours, then rewind to the common checkpoint and emit the same terminal answer. A reader may have borrowed a slice of one detour before it was abandoned. Terminal private contexts are equal. The detour samples have different action identities or values, and the

reader can have different attention inputs. The loss is a function of the full accepted trace and its reward, so terminal contexts do not determine it.

2.4 Audit as a fiber of observations

Let $\mathcal{T}$ be the category of legal trace prefixes, with one arrow for extension. An observation functor can send a trace to a record containing terminal contexts, scores, model weights, or another chosen projection. A replay witness refines the observation by lifting it to a trace with additional structure. The fiber over an observation can have many elements, as the example shows. Adding numerical scores narrows the fiber, and adding the seeded environment and deterministic computation can narrow it further. This perspective explains why partial evidence can justify a constrained claim without identifying the full generating execution.

For a trace ordering with concurrent incomparable events, a stronger provenance record can keep a partial order and attach a linearization choice. Given deterministic pure event transitions with disjoint write footprints and independent keyed draws, adjacent swaps of incomparable events preserve the value assignments. A mutable tool or global draw stream adds causal dependencies and may break that exchange rule. The reference compiler uses one concrete chronological list. Its certificates are explicit about the difference between validating that list and showing that a different schedule would produce the same samples.

3 Compiler passes and translation witnesses

3.1 Separating the controller from tensor terms

The source language has an effectful event vocabulary and a pure tensor calculus. Effects include branching, waiting, appending, mounting, and restoring. They manipulate handles and ledgers rather than real vectors. A successful transition creates or selects a typed port. The smooth calculus consumes those ports through variables, local bindings, products, abstraction, application, and typed primitives. Functions have fixed tensor shapes at the host boundary. Higher-order terms can express reusable kernels without making the controller's discrete branch choice a differentiable operation.

For a fixed accepted trace, each role has an interface. A private recurrent checkpoint represents state H and has a continuation $\mathsf{step} : H \times X \to H \times Y$. A published attention slice represents a pair of key and value tensors with native positions. A mount is an ordered selection of such slices at declared layers. A sampler consumes a logit vector and a nonempty finite support, and returns a discrete token with an action occurrence ID. The backend treats this token as fixed in the replay of its observed trace. It differentiates the probability of the chosen token with respect to the parameters through the same graph.

A precise implementation can model transitions as typed judgments $S \xrightarrow{e} S'$ with partiality for invalid events. A branch capability carries the permission to append on the branch. A private checkpoint capability authorizes restoration on that branch. A read capability authorizes access to a specific range and layer set. Capability names live in the effectful state. A smooth tensor term can take the materialized result of a read as an input, but possesses no authority to acquire new handles. The reference shares publishable appends within one episode and leaves tenant authorization at the deployment boundary.

3.2 Prefix stability

For a fixed configuration Γ, let $f_\Gamma(s, e)$ be the deterministic transition on a finite state and event. Let $R_\Gamma(s, w)$ run a list w by binding successful transitions and stopping at the first error. The state contains a permanent node list $N(s)$. A transition can push a fresh node or leave this list fixed, while it may alter the live branch map.

Theorem 3.1 (Prefix stability of accepted plans)**.** *Let uv be a legal event list and let s_0 be the initial state. Running u and then v on its saved state equals running uv directly. Moreover, the numerical nodes emitted by u form a byte-for-byte prefix of the node specification list emitted by uv, under the same canonical encoding. Each action occurrence in u retains its owner and predecessor view. Final live tails and terminal status may change.*

Proof. Induct on the length of u. For an empty list both expressions apply R_Γ to v at s_0. In the nonempty case both evaluate the same first transition. Apply the induction hypothesis to its successful state. Next inspect transition effects on $N(s)$. An append copies the current private parent and mount list into a fresh immutable descriptor. Every later operation either pushes another descriptor or leaves the list unchanged. The old token order, types, predecessor ports, and per-token support are therefore unchanged. The local rule for a first-token score refers to the immutable predecessor, and each subsequent score refers to an earlier token in its own node. These references stay fixed. Tails and closure markers belong to the mutable projection of state and can evolve independently. □

A partial rollout submits open traces and a final record checks the sealed condition. Empty prefixes are valid transition states even when the exposed evaluator requires at least one token. Configuration-dependent resource and closure checks are stated separately from the algebraic list theorem. In the Lean companion, `run_append` checks the list identity for an abstract transition, and the plan implementation follows the append-only representation. The latter correspondence is established by inspection of the transition cases, with invalid cases exercised in the reference witness.

The theorem gives a criterion for resident caching that is stronger than cache key equality on a token string. If a plan extension preserves the full numerical descriptor for its old prefix and retains the behavior version, the old node output remains a valid resident value. If a deployment changes a renderer, adapter, or numerical contract, it has changed the premise of the theorem. A restart can reuse the logical descriptor if the new backend refines its interpretation.

3.3 Predecessor views and bridge insertion

Let $S(c)$ be the view that determines the distribution available at cursor c. For a vanilla autoregressive append, the last token output of the private parent determines it. A mount updates visibility without feeding foreign tokens through the private recurrence. The parent output then remains a valid record of the old view. A forced bridge under the new view supplies a fresh output. When one wants a new distribution without an added token, the interface must provide an explicit pre-action evaluation with the same visibility and recurrence semantics. The compiler's default rule accepts the bridge form.

This precondition is observable in a trace. At a first sampled token of a node, verify that a predecessor exists and that its end position and visible mounts agree with the current cursor. If

a node begins with a forced token, later sampled tokens can use its newly computed output. A closing action may be sampled and should be scored if it is part of the policy. An administrative close or injected terminal marker has a different provenance. A decoder which labels every terminal marker as model-generated changes the episode likelihood.

A similar rule governs template translation. Let x_c be the frozen token prefix and $r(c, d)$ the incremental rendering of next turn d. Let $T(c, d)$ be a canonical branch-preserving tokenizer output. The incremental result is admissible when

$$x_c \parallel r(c, d) = T(c, d).$$

This is a token equality. It can fail even when suffix strings agree. Consider a tokenizer with one symbol for the sequence `ab`. Separately tokenizing `a` and `b` can give two symbols instead of the combined token. A renderer may avoid that boundary, keep a sufficient transducer state, or regenerate the affected prefix under a new descriptor. Template changes that strip earlier turns are more substantial and require a declared branch-preserving interpretation. This keeps token replay a checkable property instead of a presumption about text equality.

3.4 Type checking and register layout

The Lean compiler assigns base types $\mathsf{Ten}[\vec{d}]$, products, and arrows to source terms. Primitive signatures determine which operations can combine those types. A successful kernel lowering interprets tensor variables as register references, pairs as register-bearing values, and abstractions as closures over their source environments. Every primitive instruction uses registers assigned earlier. Local `let` values are emitted once. Normalization fuel bounds the reference implementation's compilation effort, and callers receive a structured error when it is exhausted.

The emitted tensor plan is an acyclic register sequence. An argument binding occupies a prescribed register range and an instruction's result follows its operands. Interpreting this sequence on real tensor values computes the term's denotation. One can prove this by induction on the successful lowering derivation, relating a register to a source value, a pair to the product of its component values, and a closure application to evaluation in its captured environment. This gives ordinary correctness for the specified primitives. A finite-precision backend is compared to that interpreter through an explicit numerical contract.

For a node list with lengths $q_0, \dots, q_{m-1}$, define

$$b_0 = 0, \qquad b_i = \sum_{j<i} q_j, \qquad I_i = [b_i, b_i + q_i).$$

A row identifier is the pair (i, j) with $0 \le j < q_i$. Its packed row is $b_i + j$. This map is injective, because distinct node intervals are ordered and disjoint and local addition is injective. It is surjective onto $[0, \sum_i q_i)$ by induction on the list. Empty nodes can be omitted from the layout and the same half-open bounds hold. In the reference compiler append nodes are nonempty.

At a layer with n visible earlier keys and a local block of width q, put the earlier keys before the local ones. For $0 \le i < q$, the accepted packed columns satisfy

$$j < n + q \text{ and } j \le n + i.$$

This is equivalent to being in the earlier prefix or having the form $n + j'$ with $j' \le i$. The Lean companion checks this equivalence and ordered nonoverlap of the node intervals. These results rule out two classes of indexing errors when a packed kernel uses the declared layout. A different attention policy can use a more general explicit mask, which then needs its own matching predicate.

3.5 A translation-validation criterion

A reference plan can be much slower than a chosen backend. The validator need not solve equivalence of arbitrary smooth programs. Instead give the backend plan a finite witness relating its row occurrences, gate boundaries, and action score sites to the accepted graph. The witness can include compiler rewrite steps with known equalities. A reused node, an inserted numerical copy, and a removed duplicate layout row may each be legal when occurrence identities and multiplicities are accounted for.

Definition 3.2 (Port-preserving translation witness)**.** A translation witness supplies an injective representation of logical token occurrences, a matching between logical ports and target values, and the operation-level equalities used for each mapped gate. It preserves the ordered arguments of recurrence, keys and values with their native positions, attention visibility per query, and the predecessor view and support of every sampled action. A target value may be a view of a shared allocation or the output of a local subprogram.

Theorem 3.3 (Forward and reverse refinement)**.** *Suppose the target realizes a port-preserving witness for a fixed trace on a parameter open set. If the witnessed gate equalities hold as smooth maps on that set with all numerical dependencies live, the decoded target outputs and the reference outputs are equal as smooth functions. Their vector-Jacobian products coincide. The target score and its derivative coincide with the reference when the same sampled occurrences and objective weights are selected.*

Proof. Induct on logical gate order. A source port is related by the witness. Every other gate gets its ordered inputs from established predecessor relations. The gate equality then establishes the output relation. Thus decoded outputs agree for all parameter values in the open set. Differentiating this functional equality yields the pullback equality. The score evaluation selects the same output row, finite support, and token value for every sampled occurrence. Summing the same differentiable score terms with the prescribed weights proves the last claim. □

The liveness condition matters in an autodiff implementation. Two programs can return identical arrays on the same numerical inputs while one uses a stop-gradient on a borrowed tensor. A primal test at that point will accept both. A pullback witness for the read must establish that the consumer cotangent reaches the producer with the right range, even if the borrowed tokens carry no policy loss at the reader. Conversely, an edge used only as an external historical observation can have a declared constant semantics and a zero pullback. This choice belongs in the provenance of the replay plan.

3.6 Interpreting a fused kernel witness

The original architecture uses fused projections and GPU kernels for different layer families [5, 6, 7]. For tokenwise projection a witness gives a row permutation and a common linear map, so

permutation commutes with projection. For private recurrence it identifies the initial checkpoint and convolution tail for each segment. For sparse experts it gives the expert assignment, capacity convention, and scatter back to the original row. For rectangular attention it identifies the exact ordered prefix and suffix. These witnesses are local enough for differential tests and strong enough to support the functional proof when the library satisfies its mathematical contract.

A borrowed range may cross pages or be represented by multiple subviews. Their total logical order and multiplicity must match the reference. A streaming softmax can combine per-block maxima and normalized exponential masses. Exact arithmetic then gives the same output as one concatenation. Under floating point, the merge order and dtype can alter the result slightly. Numerical audits report tolerances and keep track of the relevant backend, precision, and mode. A checker can compare a fused implementation with a streaming or dense reference on the same graph without replaying any policy draws.

4 What a replay certificate can establish

4.1 Integrity and provenance fields

An audit takes a trace envelope, a compiler profile, and an expected claim. It returns a verdict and the quantities needed to interpret it. For a standalone step record, the envelope includes the run coordinates, source and compiler fingerprints, behavior version, groups and their rewards, and the full chronological events. The output includes a raw-file digest, the digest of the ordered compiled plans, an ownership count, and the result of each selected check. In a cloud deployment a manifest also names its durable input versions and the expected contributing ranks.

The evidence structure can be described by an acyclic provenance graph of artifacts and activities, in the sense of the W3C provenance data model [8]. A rollout consumes behavior weights and produces event logs. Compilation consumes those logs and produces typed plans. Replay consumes the plans and current weights and produces an action score table and gradient messages. Reduction consumes an expected message set and produces a gradient. A checkpoint activity consumes that gradient and an old optimizer state and produces the proposed next state. Keeping separate provenance nodes for these activities allows one to check which operation a digest or signature names.

A raw SHA-256 digest of the file commits to its bytes relative to an expected value from a trusted source. It detects a mismatch under that premise. A digest computed after receiving an untrusted file is a label for the received file. Provenance authentication can use signed statements binding the operation, subject digests, producing identity, configuration and run coordinates, in the style of software supply-chain attestations [9]. The trust policy must say which principal may assert each relation, and external tools may have different authorities from training workers.

Canonicalized digests support semantically stable interchange only when serialization rules are explicit. JSON object key order and numeric spellings can differ. Duplicate object keys can give different values under two parsers. The reference auditor rejects duplicate object keys and non-finite JSON constants. It hashes raw records and separately hashes a compact, sorted-key ASCII encoding of compiled plans under a named profile. Domain tags distinguish one plan from an ordered plan list. The reference profile is specified by the delivered Python serialization. RFC 8785 is a separate cross-language canonicalization scheme and is a suitable option for a wider

interchange contract when its number and string rules are applied consistently [10].

4.2 Structural and numerical audits

Structural validation replays the admitted transition list and rejects out-of-range slices, references to future publications, missing private predecessors, and illegal support declarations. It can count actions on abandoned paths by traversing all final private ancestries and subtracting their owned occurrences from the full ledger. Mounted ranges may come from the complement of that set. A counter of abandoned actions does not by itself determine which abandoned numerical values remain reachable through borrowed edges.

Numerical audit uses the recorded behavior weights θ_0. Each action ledger record supplies a predecessor row z, ordered support S, and observed token $x \in S$. Its recomputed log-probability is

$$\operatorname{logp}(z, x, S) = z_x - \log\left(\sum_{j \in S} e^{z_j}\right).$$

The audit compares this quantity with the stored value. Tokenizer, logit transforms, temperature, and support filters are part of the policy contract. If support was itself selected by a routing or masking rule, a replay of the declared support checks the score conditional on that support. A stronger claim also requires validating how the support was produced.

Stored probabilities should survive round trips through the record format. The Lean reference keeps the exact parsed JSON number rather than reformatting it through a binary float. It checks finite range and sign before accepting the action record. The Torch comparison rounds to the declared tensor dtype. This distinction prevents a textual formatting step from becoming a source of mismatch and makes a tolerance attributable to the numerical comparison.

Proposition 4.1 (A selected score does not determine the policy)**.** *Agreement of recorded log-probabilities for the selected actions of one trace gives no general bound on the KL divergence between the two full action distributions.*

Proof. Use three actions and select the first. Let the distributions be $(1/2, 1/2 - \alpha, \alpha)$ and $(1/2, \alpha, 1/2 - \alpha)$ with $0 < \alpha < 1/4$. Both assign exactly the same probability to the observed first action, while their KL divergence is $(1/2 - 2\alpha)\log((1/2 - \alpha)/\alpha)$, which diverges as $\alpha \to 0$. The example applies even with identical full support. □

A research-scale validator can compare complete conditional distributions when they are available, or collect statistically valid samples for estimating an expected log-ratio with units declared. The reference reports the maximum discrepancy on sampled action scores for a fixed record. That quantity is directly related to the behavior denominator used by the training surrogate.

4.3 A tolerance has an objective-level meaning

For an action a, let ℓ_a be the true behavior log-probability under the declared replay contract and $\hat{\ell}_a$ its stored or approximated value. For current log-probability c_a, the exact and approximate ratios are $r_a = e^{c_a - \ell_a}$ and $\hat{r}_a = e^{c_a - \hat{\ell}_a}$. Assume $|\hat{\ell}_a - \ell_a| \le \delta_a$. Define a clipped gain

$$F_A(r) = \min\{Ar, A\ \operatorname{clip}(r, l, u)\}, \qquad 0 < l < 1 < u.$$

Theorem 4.2 (Replay drift and gain stability)**.** *For every owned action,*

$$e^{-\delta_a} \le \frac{\hat{r}_a}{r_a} \le e^{\delta_a}, \qquad |F_A(\hat{r}_a) - F_A(r_a)| \le |A|\, r_a(e^{\delta_a} - 1).$$

The ratio of complete trace likelihood-ratio products lies in $[\exp(-\sum_a \delta_a), \exp(\sum_a \delta_a)]$*. For weighted clipped gains with nonnegative weights, the total error is at most the sum of the per-action bounds with those weights. If the exact log-ratio lies farther than* δ_a *from its active clipping boundary, it stays in the same clipping region under the audit error.*

Proof. Subtract the log ratios and exponentiate the resulting interval. This gives the product bound by adding log errors. For $A \ge 0$, $F_A(r) = A\min(r, u)$, and for $A < 0$, $F_A(r) = A\max(r, l)$. Both functions are $|A|$-Lipschitz. Also $|\hat{r}_a - r_a| \le r_a(e^{\delta_a} - 1)$. Multiplying by the weights and applying the triangle inequality proves the gain bound. The clipping boundary is a threshold in log-ratio space, so a perturbation strictly smaller than its distance leaves the region fixed. □

At the behavior version the exact ratio is one. If $\delta_a < \min(-\log l, \log u)$, an audit error alone cannot move that action outside the unclipped band. This gives a principled relation between a replay tolerance and a trainer's clipping width. A length-normalized average can keep this per-action error scale when the number of owned terms grows, while a joint likelihood ratio compounds the bound over the episode. A tolerance suitable for a local PPO-style check may be too large for exact trajectory reweighting.

The bound concerns recorded denominators at fixed current scores and objectives. It does not certify gradients when the backend applies a different pullback to the same values. When the current map and its derivative are the same and both ratios stay on an active smooth linear branch of F_A, the derivative error is bounded by $|A|r_a(e^{\delta_a} - 1)\|\nabla c_a\|$ by the chain rule. At a clipping boundary, an arbitrarily small error can change which derivative branch is chosen. This is why the auditor treats numerical score agreement and reverse-port evidence separately.

4.4 Seeded regeneration

A seeded replay supplies a stable key for each decision. One convenient key is the tuple of run seed, step, logical rank or shard, prompt group, episode, branch capability, and decision label. Deterministic actions draw nothing. Counter-based or keyed generators let independent scheduling orders recover the same draw for the same occurrence, as in parallel random-number generation [11]. To recover the selected token, one must also have a deterministic sampler, support order, and numerical model at that occurrence.

The reference derives seeds by hashing labelled tuples and uses a recorded PyTorch profile to sample a categorical distribution on a finite support. Its numerical and generative audits use the recorded Torch version and device kind. Code and compiler fingerprints are reported separately, so a rebuilt compiler can be allowed under a stated profile. A CPU run can match choices on the same software and weights without making a claim of reproducibility across all GPUs or different sampler implementations.

Proposition 4.3 (A decision margin for common uniforms)**.** *Consider inverse-CDF sampling on a fixed ordered support using the same* $u \in (0, 1)$ *in two evaluations. Let* F *and* $\tilde{F}$ *be the cumulative distributions. If* $\max_j |F(j) - \tilde{F}(j)| \le \eta$ *and* u *has distance greater than* η *from every internal boundary of* F*, the chosen action is unchanged.*

Proof. Every boundary remains on the same side of u, so the first cumulative probability greater than or equal to u has the same index. □

An analogous argument can be given for Gumbel-max sampling in terms of the gap between the highest and second-highest perturbed logits. Such conditions quantify why small numerical deviations often preserve sampled actions but cannot guarantee it at a decision boundary. The companion's regenerated rollout comparison is an observed equality of action sequences and rewards. Its random-number and kernel profile determines the reproducibility scope.

4.5 Certificate checking order

The useful audit order is structural validity, numerical replay, seeded regeneration, then publication of the audit result with provenance. Signature or manifest verification may precede any expensive parsing when the trust model requires it. Structural replay can reject malformed work before allocating accelerator tensors. Numerical replay can operate on a recorded episode without sampling. Seeded regeneration tests the controller and environment as well as the numerical model. A certificate should state which stages completed. If a tool transcript is unavailable, the structurally validated and numerically replayable subset can still support a narrower claim.

Resource limits belong at the boundary of an implementation that accepts untrusted logs. A maximum event count, native-position cap, and token budget bound parts of the logical plan. They do not bound serialized parser input size, process spawning, or the number of times an attacker may submit. The companion is a local artifact verifier. A public admission service would need a byte limit before decoding, bounded parser depth, process isolation, and per-principal quotas. Such controls constrain the checker, while the certificate predicates describe the semantics of accepted records.

5 Reconstructable cuts and selective evidence

Full episode logs can contain prompts, tool results, and model artifacts with different access policies. An audit interface should be able to state what evidence is necessary for a particular computation. A useful unit is a dependency cut with typed boundary values. It can represent a task whose preconditions are already verified elsewhere, a subgraph executed on another accelerator, or a part of the episode disclosed to a reviewer.

5.1 Forward evidence across a cut

Let G be a finite typed dataflow graph with deterministic gates. For a subset U of gates, the incoming cut $\partial^{-}U$ consists of edge occurrences whose producer is outside U and consumer is inside U. Expose the input type, canonical edge identity, and value of each cut port. For retained gates, expose their primitive contracts and all edges between them. A graph with empty incoming cut contains all dependencies of its gates and can be replayed from global inputs that occur inside it. Other subgraphs can use cut values as additional boundary inputs.

Proposition 5.1 (Replay on a typed cut)**.** *If incoming cut values are the original outputs of their declared producer ports, evaluating the induced subgraph on those values reproduces the full*

evaluation restricted to its gates. If each cut value is supplied with an incorrect value, the same restricted evaluation instead computes the function of the supplied boundary inputs.

Proof. Take a topological order on the retained gates. Every incoming argument either has a producer earlier in this order or is supplied by the cut. In the first case its value agrees with the full evaluation by induction. In the second it agrees by assumption. The local gate is deterministic, so its output agrees. With alternative cut values, the same evaluation is the ordinary parameterized interpretation of the subgraph with the new inputs. □

This yields two audit tasks. A boundary-conditioned check verifies the subgraph given cut values. A closed check verifies those values by providing the transitive dependency closure or a separate authenticated proof. The former is appropriate when a trusted tool supplies a result. The latter is appropriate when the goal is to audit a borrowed slice computed by an untrusted worker. The set of disclosed inputs must include provenance and positional information whenever they affect the subgraph, even if their numerical types alone do not expose this requirement.

5.2 Reverse evidence across a cut

For a subgraph used as an open function $f : X \to Y$, a verified forward witness gives $f(x)$ at the relevant inputs. A reverse audit also needs a vector $\bar{y}$ at each outgoing port and its mapping into the full loss computation. Its local reverse result is $Df(x)^*\bar{y}$. If several outside edges read the same output, $\bar{y}$ is the sum of their cotangents with their logical multiplicities. Taking a subgraph's derivative with a zero boundary cotangent is a different loss even when its forward values match the original.

Theorem 5.2 (Cut composition of pullbacks)**.** *Partition a finite acyclic smooth graph into regions so that the directed quotient of its cross-region dependencies is acyclic. Suppose each region supplies its forward values on typed cut ports and its local pullback. Process the regions in forward topological order and reverse them for cotangent propagation. Summing each cut incidence once gives the same full pullback as evaluating the graph without the partition.*

Proof. Forward equality follows by applying Proposition 5.1 to each region when its cut arguments are available. The function computed by gluing the regions is the composite of their functions with products, projections, and diagonals for shared values. The chain rule transposes this composite in reverse order. The transpose of a diagonal adds the contributions of all its incidences. This is precisely the stated reduction, so the resulting pullback is the full one. □

The acyclic quotient premise is about the chosen placement regions. Arbitrary partition of an acyclic graph can produce a cycle after contraction, for example when a region contains both an early and a late gate with another region in between. Such a placement can still use multiple ordered phases per physical worker. The logical regions for the theorem must be chosen at that finer grain. Separating worker identity from phase identity prevents a false cycle in the dependency manifest.

5.3 Membership commitments and coverage

A manifest can commit to an ordered list of event or contribution records. An ordinary Merkle tree, with different domains for leaves and internal nodes, permits a disclosed record and its position to be checked against an anchored root with a logarithmic-size path for a balanced tree. Transparency log designs provide a precise precedent for these inclusion proofs [12]. The implementation here hashes complete local records and ordered plans. It does not implement a network transparency log.

Inclusion of the disclosed records is a distinct question from completeness of a replay. A proof that a gradient message occurs in an authenticated list leaves open which other messages were expected. The verifier needs a trusted expected-key set derived from the plan or a completeness witness. For a cut audit, this set is all incoming and outgoing incidences for the stated verification scope. For a committed optimizer update, it is all required data-parallel groups or reverse contributions. An auditor that checks inclusion only can accept a subset chosen to hide difficult cases.

Stable roots and version IDs also reveal equality relationships across releases. A system that allows low-entropy tool results to be guessed from public deterministic digests has a different confidentiality contract from one that keeps the record private and reveals only authorized proofs. Content hashes establish identity relative to a profile and an expected digest. They do not allocate read authority. A deployment can bind access decisions to capabilities and separate internal addressing from public audit evidence. The local reference uses one trusted episode scope and exposes its synthetic records for reproducibility.

5.4 A lower bound on reconstructing discarded choices

Suppose an observation O is the final model state together with the visible private histories. Let there be M admissible full event traces with that same observation. If an additional fixed-width tag permits exact reconstruction of every one of the M traces from O, the tag uses at least $\lceil \log_2 M \rceil$ bits. This follows from injectivity of the tag on the fiber. The lower bound is independent of the compressor or storage layout.

One way to obtain a large fiber is to sample k binary detour actions, then rewind to the original cursor and emit the same final answer. With both actions in support at each detour, all 2^k paths can share the same final private history. If their training advantage is zero, they can also share the same optimizer state under a deterministic update. The additional evidence still needs k bits in a fixed-width worst case. When the seeded generator and complete environment are known, a seed can compress many choices by restricting the fiber to a smaller family. The cost then shifts to retaining the versioned procedure and its input dependencies.

This observation gives a design principle for selective evidence. If the question is only which state a worker resumed from, the committed checkpoint and its provenance can suffice. If the question is whether a rewound decision contributed to an update, its occurrence or an equivalent replay witness must remain available. If the question is whether that decision could have changed a reader's behavior, the borrowed dataflow and the intervention semantics also matter.

6 Recoverable updates across workers

A checkpoint answers two different questions. It gives the optimizer state from which a worker can continue. It can also certify that a particular logical update consumed a particular set of episodes. The second use requires a manifest and a version transition. We state the full protocol for a distributed executor and distinguish it from the shared-filesystem worker in the companion.

6.1 Typed contributions and deduplication

Let a logical update have an identifier u and a sealed predecessor model version. For every required reverse incidence, define a key

$$(u, \text{episode}, \text{layer}, \text{producer port}, \text{consumer port}, \text{occurrence}).$$

The final field distinguishes repeated views of one slice used as separate inputs. Expected keys are derived from the compiled graph and the declared schedule of work. A producer's incoming cotangent is a sum over these keys. If a consumer sends a coalesced block, its manifest must specify a disjoint subset of keys and the reduction convention. For data-parallel parameter gradients, a key instead identifies a disjoint logical group or shard and its weighting rule.

A receiver stores a partial map from expected keys to validated contributions. A repeated key with the same content is a retry. A repeated key with incompatible content is a conflict, and a receiver can stop the update pending resolution. A conflicting message has no well-defined merge in this contract. A complete receiver reduces one contribution from each expected key. Over exact reals, the result is independent of arrival order. A finite-precision contract fixes a stable reduction tree if bitwise reproducibility matters.

Proposition 6.1 (Idempotent contribution union)**.** *Let K be the finite expected key set and suppose accepted partial maps agree on overlaps. Their union is associative, commutative and idempotent. If its domain is K, reducing one value per key gives the declared complete sum.*

Proof. Agreement makes the union a well-defined function. For any key present in several maps, they all assign one common value. Evaluating a union on a fixed key therefore has the same result under regrouping, permutation, or duplication. Once all keys are present, each of their prescribed values occurs once in the sum. □

This argument treats validated message contents as inputs. It does not prove that an arbitrary contributor calculated the intended value. The latter is supplied by the execution or attestation model and by the port-preserving witness. A stronger auditor can regenerate a contribution with its numerical closure, compare it to a declared tolerance, and bind the comparison to its key and version. Receivers distinguish the producer's identity, the job's logical incidence, and the transport attempt identifier.

For a graph split across workers, topological induction reconstructs the forward values from declared inputs. Reverse induction applies Proposition 6.1 to the expected outgoing dependency ports of each producer. A worker can release its local activation after making all its expected outputs durable or reconstructable. If one predecessor is reconstructed under a different version, the numerical argument fails at that gate even if all message IDs are syntactically valid.

6.2 An observable commit protocol

A durable store supports a linearizable compare-and-swap on a run head [14]. The head points to an immutable checkpoint descriptor with a model digest, optimizer digest, logical update ID, parent checkpoint digest, and complete contribution manifest. An updater computes a proposal from the head it read. It writes the proposal and all objects reachable from it before attempting to publish. The conditional write changes the head only if the old pointer and any required fencing epoch still match.

Suppose an updater loses the acknowledgement after sending the conditional write. It can inspect the head and the retained parent chain, or a durable index of committed update IDs. A commit that is already an ancestor is still visible as a completed step. A retry may reuse the same immutable proposal when the expected head is still current. If a different logical step has advanced the head, applying the old proposal would use a stale optimizer state. The conditional write fails, and the scheduler must determine whether the old work is obsolete or should be translated into a new step with new inputs.

Theorem 6.2 (Single visible successor)**.** *Assume a linearizable head, immutable complete checkpoint objects, and a valid initial descriptor. A commit may move the head only to a proposal whose parent equals the current head and whose update ID has no earlier visible occurrence. Then the visible checkpoint sequence is a chain, any parent has at most one visible immediate successor, and retrying a completed logical update cannot insert it twice. If each published proposal uses one valid contribution per expected key, its optimizer state is the stated update of its parent.*

Proof. Linearizability orders all successful compare-and-swaps. The first attaches to the initial head. Inductively a new success attaches to the exact head after the preceding success, so the visible sequence extends one chain. Two successful writes with the same expected head cannot both attach because the first changes it. The update-ID check excludes reinsertion on a later head. This check can use the immutable ancestor chain of the expected parent before the compare-and-swap, because any intervening change of parent makes the write fail. The final assertion follows from the complete reducer and the stated optimizer rule. □

Crash-free completion requires availability of the state store and eventual delivery or reconstruction of all expected inputs. The theorem is a safety property under retries. It permits repeated compute, duplicated message attempts, and unreachable staged objects. Garbage collection can delete a staged object after it has been made unreachable and after a grace period for in-flight valid commit attempts. Visible ancestors are retained according to the audit and recovery horizon.

A process-local POSIX rename can give atomic visibility for a single writer on an appropriate filesystem. It lacks the concurrency premise of the distributed head protocol. The reference worker synchronizes its checkpoint file and containing directory and expects exclusive use of the checkpoint path. Its distributed launcher design gives rank zero the writer role. Multiple jobs sharing one path would require the external fencing and compare-and-swap contract.

6.3 Failure cases

Observed failure	Durable evidence	Recovered behavior under the contract
Producer crashes before sending a gradient	Expected key absent	Recompute its closure under the sealed version and send the same logical key.
Message acknowledgement is lost	Key may already exist in the reducer	Retransmit and compare content on duplicate admission. The contribution counts once.
Worker loses a resident KV replica	Descriptor and predecessors remain available	Rematerialize the typed tensor from its recorded plan and version.
Updater crashes before publication	Previous head and perhaps staged objects	Verify completeness and retry the conditional write, or collect orphaned stages.
Updater crashes after publication	New head or a later head with the update in its ancestry	Read the committed version and continue from that state.
Two updaters race on one old head	Competing staged descriptors	One conditional write succeeds. The loser checks ancestry and discards obsolete work.
Tool request times out	External operation might have occurred	Reuse the logical effect key or query its durable receipt when the tool supports that contract.

Distributed snapshots record consistent cuts of computation and communication state [13, 15]. The present recovery design uses a sealed finite dependency graph to derive which logical work must be complete before publication. It can rebuild intermediate values from their provenance. An asynchronously expanding graph instead needs an explicit rule for sealing the final expected set. Counting whichever messages have arrived at a timeout silently changes the optimized objective.

6.4 External effects and compensation

A rewind changes the internal cursor. It does not undo an external write already accepted by a tool. Associate a distinct idempotency key with every logical external effect and store a request descriptor before the call. If the tool records the key and its result durably, the worker can safely retry or retrieve the result. If an effect is intentionally compensated, record a separate compensating operation and its authorization. The first operation remains a fact in the event log.

There is a simple limit to what a training service can prove without tool support. Consider two executions indistinguishable to the service after a lost reply. In one, the tool received and performed the operation. In the other, it never received it. A blind retry can duplicate the first effect, and abstaining can leave the second absent. The service cannot choose between these outcomes based on its own state. A durable idempotency record, status query, transaction participant, or application-specific compensation changes the information available. The immutable trace can state which of these mechanisms is in use.

6.5 Data-parallel training and placement

Let G_r be the set of prompt groups processed by logical rank r, with group losses L_g and nonnegative weights w_g. The global gradient of the weighted mean is

$$\frac{\sum_r \sum_{g \in G_r} w_g \nabla L_g}{\sum_r \sum_{g \in G_r} w_g}.$$

A mean of local means has this meaning when the ranks have equal total weight or when a further weighted reduction corrects for unequal weight. A rank retry should preserve its logical group IDs, not create a fresh group with a different random draw. Physical placement can change without changing the episode set when logical shards and rank mappings are separate. In the reference launcher rank is part of the seed profile, so changing world size changes the declared experiment. Resume requires the same world size.

Within an episode, pinning a foreign slice on the same device can use a read-only view. Across hosts, the bytes have to be reachable under an explicit transport or remote memory contract. Keeping one content ID for several replicas preserves semantic identity while allowing different physical costs. A scheduler can optimize the placement of contiguous strips and borrowed views under memory budgets. The audit record gives a way to check that its decision preserves logical visibility and action ownership.

For resource accounting, charge private token work, distinct physical allocations, communicated bytes, and pending reservation separately. A producer can have many readers of one cached allocation and many numerical contribution keys. The logical key count bounds reducer work while storage is governed by live pages and in-flight transfers. A weakly consistent reference count is unsafe for reclaiming the last pin unless reclamation is delayed until every active reader's lease has expired or been fenced. The reference store retains all per-episode nodes and avoids this reclamation problem at the expense of memory. A production design can use ownership epochs or explicit pin lifetimes with versioned handles.

7 The update ledger and reproducible reductions

A replayable episode is one input to an optimizer update. An update also depends on which episodes were grouped, how advantages were assigned, and which finite-precision reduction combined their contributions. The update ledger records those facts. It can be checked independently of the numerical storage layout used by the training process.

7.1 Sealing a group before replay

Let a behavior prompt group have $m \geq 2$ episodes, with rewards $R_1, \ldots, R_m$ and sampled occurrence sets $\mathcal{A}_i$. Its leave-one-episode-out advantages are

$$A_i = R_i - \frac{\sum_{j=1}^{m} R_j - R_i}{m - 1}.$$

An update descriptor commits to the ordered group members, their reward sources, a policy snapshot, and a normalization rule. The score tables for distinct episodes may be computed

concurrently. The group advantage must use the complete manifest. A missing episode can change the score weights of every remaining episode even when their own replays agree.

Proposition 7.1 (Group deletion changes other weights)**.** *For $m \geq 3$, fix an episode i and write B_i for the mean reward of the other episodes. Deleting a different episode k changes that baseline by*

$$B_i' - B_i = \frac{B_i - R_k}{m-2}.$$

Thus a record set missing one reward can pass all remaining per-action replay checks while representing a different update.

Proof. After deletion, the other rewards sum to $(m-1)B_i - R_k$, and their count is $m-2$. Subtract B_i from the resulting quotient. The score calculations within the retained episodes use the same model, inputs, and supports independently of this change. □

As a concrete example, four rewards $(1, -1, 1, -1)$ yield advantages $(4/3, -4/3, 4/3, -4/3)$. Deleting the final episode gives the remaining advantages $(1, -2, 1)$. If an available subset is intentionally used, a ledger can describe that new experiment by publishing a new group identity. Substituting it for the sealed group changes the estimand. In the companion, rank-local groups are complete by construction. The auditor checks their cardinalities against the recorded configuration and recomputes rewards during seeded regeneration.

A group split across ranks can use an integer count and reward sum as sufficient statistics for the baseline. Each disjoint shard reports its count and sum with a coverage manifest. The aggregator combines them, then broadcasts the total and verifies the expected count. Each shard computes its local advantages with the global total. With rational or exact integer rewards where applicable, these statistics can be reduced exactly before converting to the optimizer dtype. Identity still matters, since equal counts and sums can hide replacement of one episode with another. The manifest binds the statistics to the members.

7.2 Separating update completeness from numerical equivalence

An update witness comprises a membership ledger, an objective descriptor, a reduction witness, and a model-state transition. The membership ledger gives the episode and group keys, action-count denominators, and behavior version. The objective descriptor specifies clipping, temperature, reward weighting, and gradient accumulation steps. The reduction witness gives the mapping from contributions to the global mean, including the order if needed. The transition names the optimizer state before and after the update. An auditor can test each claim at the granularity at which it is made.

Example 7.2 (Equal policy replay with different optimizer state)**.** Two workers replay the same group with the same current parameters and obtain the same gradient g. An adaptive optimizer can have different momentum or second-moment accumulators even if the model parameters agree. Its next parameters can differ on identical g. A model digest and gradient digest alone therefore cannot determine an optimizer transition. The parent checkpoint must include the optimizer state and step count.

The equality of model, optimizer, and next-step fields in the companion's restart check is stronger than equality of the last checkpoint's model weights alone. It remains a check at a fixed deployment profile. If a worker loads a checkpoint under a different arithmetic or sampler profile, numerical agreement becomes a new claim. Likewise, replaying one optimizer microstep requires the exact order in which its groups or gradient shards were presented if the update algorithm makes that order significant.

7.3 Finite-precision reduction as a witnessed choice

For a scalar coordinate of gradient contributions, let $g_1, \dots, g_n$ be the exact values presented to the reducer. Assume no overflow or underflow and a relative rounding model in which each addition is multiplied by $1 + \delta$ with $|\delta| \le u$. In a fixed reduction tree of maximum depth d from an input leaf to the root, the computed sum has the form

$$\widehat{G} = \sum_{i=1}^{n} g_i(1 + \eta_i), \qquad |\eta_i| \le \frac{du}{1 - du} \quad \text{for } du < 1.$$

To see this, distribute the multiplicative error at each internal addition across both of its subtrees. Each leaf acquires a product of at most d factors in $[1 - u, 1 + u]$. The maximum deviation from one is bounded by the displayed expression, using $(1 + u)^d \le (1 - du)^{-1}$ and $(1 - u)^d \ge 1 - du$. The absolute error is therefore at most $\frac{du}{1-du} \sum_i |g_i|$. A balanced tree has logarithmic depth. The estimate concerns the reduction of supplied coordinates and does not include errors in their computation.

A retry that receives the same validated key can leave the reduction tree unchanged. A reduction that instead adds in arrival order may return slightly different values after a retry or reassignment even if the same mathematical sum is specified. The record system should distinguish a claim about the exact-real reduction, a bound on observed error, and a fixed-bit comparison. The single-head commit can publish any of these profiles if the admissibility rule states the expected one. Different profiles can be compared with the same sealed membership ledger.

7.4 What can be regenerated on demand

Storing all numerical activations would make a replay simple but expensive. A checkpoint-recompute strategy stores a sealed plan and a set of versioned boundary activations. On audit, close the required cut under dependencies until it reaches an available boundary. Recompute under the declared kernels and verify the selected outputs and cotangents. A report can state how many gates and cache bytes were read to verify the audited loss terms. This turns the cost of a stronger evidence claim into an explicit resource quantity.

The immutable plan and behavior weights allow recomputation of probability denominators, while the update ledger decides how those denominators entered a particular surrogate. Regeneration of the controller needs the environment inputs or a deterministic environment implementation. Recovery of an externally visible side effect needs its separate tool receipt. These requirements can share the same occurrence IDs without sharing retention policy. A cloud system can then retain richer evidence for risky or exceptional updates and use population sampling for routine work under Section 8.

8 Auditing a large collection of updates

A production run may produce too much evidence to regenerate every episode. The pointwise checker remains the same, but the claimed result changes when it is run on a subset. We give an audit sampling rule that bounds the chance of missing a widespread failure after a fixed set of records has been committed. The rule applies to structural, numerical, or generative predicates with a declared checking procedure.

Proposition 8.1 (Detection in a committed population)**.** *Suppose a sealed population has N records and exactly M of them fail a deterministic audit predicate. Draw k distinct indices uniformly without replacement, independently of the population's contents. The probability of observing no failures is*

$$\frac{\binom{N-M}{k}}{\binom{N}{k}} \le (1 - M/N)^k,$$

with the numerator interpreted as zero if $k > N - M$.

Proof. There are $\binom{N}{k}$ equiprobable subsets, and $\binom{N-M}{k}$ contain only passing records. For $k \le N-M$ the quotient is a product of factors $(N-M-i)/(N-i)$ for $0 \le i < k$. Each is at most $(N-M)/N$. The zero case is immediate. □

If a claimed error rate is at least η, then k selected after commitment has miss probability at most $(1-\eta)^k$. An audit with no failures can therefore rule out prevalence at least η at a chosen one-sided significance level for this sampling design. It does not certify each uninspected episode. If the producer sees the selection before finalizing its evidence, the independence premise fails. One can freeze a manifest root and draw the sample from a separately chosen audit seed after the freeze. The root fixes the population against which missing or replaced records are detected.

Failures can cluster by model family, device generation, tool, mount direction, or number of discarded nodes. Sampling uniformly over the full population can miss a small but important stratum. A stratified audit fixes a per-stratum population manifest, samples uniformly in each, and reports coverage and failure rates for those populations. If random draws are independent across strata, the probability of missing all failures factors across strata. Without that independence, a conservative familywise bound can use the sum of the separate miss probabilities for individually declared targets.

The audit unit should match the claim. Sampling actions from a batch can detect local denominator drift. It need not detect an incomplete group ledger because it conditions on the supplied batch. A group-level audit includes its declared number of episodes, their individual rewards and supports, and the leave-one-episode-out weights. An update-level audit checks contribution manifests, rank weighting and the parent checkpoint. A cut-level audit tests specific borrowed or recurrent edges. The record system can support all these units while sharing trace and artifact digests.

8.1 Measuring the audit and its blind spots

For a numerical batch with per-action errors δ_a, report a maximum over all inspected occurrences, a high quantile with its unit, and an episode-level sum if a joint likelihood comparison is intended.

Averages can conceal a systematic error on the first action after a mount or rewind. A report should include those boundary roles explicitly. If numerical differences are driven by underflow or reduced precision, a staged comparison at a reference dtype can help localize the issue. A successful comparison in that dtype narrows the failure to a later interpretation or precision choice.

An audit of complete probabilities can measure divergence at fixed histories. A generative audit instead examines whether a controlled decoder chooses the same next action. The two tests can fail separately, as a small logit shift near a sampling threshold can change a trajectory and a larger shift away from all sampled thresholds can leave choices unchanged. Proposition 4.3 gives a deterministic decision-margin criterion for an inverse-CDF profile. The actual runtime may use a different sampler and need a corresponding stability condition.

Audited outputs are also data for debugging. A report should include the first failing logical occurrence, the two predecessor descriptors, the current and behavior support and scores, and the backend profiles. These fields can identify a missing forced bridge or a detached mount without requiring a full dump of unrelated contexts. The cut construction in Section 5 gives the exact dependencies that must accompany such a report when another party is expected to reproduce it.

8.2 Integrating a backend incrementally

A production integration can be split into small contracts. First implement the controller event recorder and validate stable prefix identities under fork and rewind. Then add an adapter for the model's complete checkpoint and tokenizer semantics. Compare fixed traces for the same layer masks before enabling sampled rollouts. Next replay grouped losses with both normalized and unnormalized objectives to confirm the scoring ledger. Finally qualify the data-parallel launcher, retention rules and distributed publication separately. This ordering follows the dependency of the claims. It avoids interpreting a difference in benchmark accuracy as the first sign that the training computation changed.

The source's pure-attention models need no recurrent state checkpoint. Its hybrid architectures require their exact recurrent state and the private causal convolution tail, including after rewinds. Sparse expert routing needs a dispatch contract because batching can change capacity or routing even at equal input tokens. Native rotary positions and full- or sliding-window visibility are part of the attention interface. Each adapter can be verified against fixed logged inputs independently of the toy implementation's particular recurrent equations. This makes the artifact a reusable validation design rather than a requirement to use one model family.

9 Reference implementation and evaluation

9.1 Artifact layout and environment

The companion includes the Lean sources for the smooth term checker, branch transition compiler, and machine-checked layout lemmas. The compiler executable `branchc` emits JSON. The Torch code provides an incremental resident evaluator, a packed evaluator, grouped reinforcement learning, and an offline audit tool. The artifact also includes a Docker build definition, a Kubernetes GPU Job manifest, and the small CPU evidence records. The compiler and trainer source are shared with the companion manuscript. The replay auditor, mutation checks, and layout lemmas are the additional implementation in this paper.

The CPU configuration uses Lean 4.19.0, Python 3.12, PyTorch 2.7.1 CPU, one rank, and a fixed seed of 17. It has a twelve-token vocabulary and hidden width twelve. The model uses a recurrent block below two attention layers, one selected retrieval layer, and another recurrent block above attention. In the training run the attention projections use low-rank factors of rank two. The optimizer uses two updates from each sealed batch, with two groups of four episodes per behavior step. These choices make the experiment a reference witness for replay and gradient provenance. They do not model the parameter scale or context limits of the source study.

The task has a bit hidden from the root. A scout can receive it as a forced observation. A reader can take a detour and rewind. It can access the scout through a selected attention layer. The root can delegate, wait, and use a transitive mount before guessing the bit. The scorer gives $+1$ for a correct guess and -1 otherwise. The source model's retrieval corpus is unnecessary for this witness, since the relevant claim is that all numerical and sampled dependencies of a small latent-read program survive a trace replay.

9.2 Audit procedure

The trainer emits one record for a step and rank. Its provenance includes the hash of the executed compiler, the runtime source fingerprint, the training options, Torch version, and device kind. The offline auditor reads the step record from disk and recompiles each episode. It counts action occurrences, mounts, and rewinds and produces an ordered-plan digest. Given behavior weights, it independently computes the recorded probability denominators with packed replay. When seeded regeneration is requested, it calls the behavior controller with the recorded seed and compares the resulting events and rewards. In update-verification mode it takes the already compiled groups and their logged rewards, reconstructs the optimizer from the predecessor state, replays the declared optimizer steps, and compares the result with the proposed checkpoint.

For the first step, the auditor regenerates the initial model from the fixed seed and constructor. For the second, it loads the preceding saved checkpoint and checks that the checkpoint's next-step index and provenance match the record. The fixed input is the logged trace, so the numerical audit can detect an altered score independently of whether the stored reward agrees with the environment. The generative check supplies that additional relation. It uses the same controller source in a fresh process. It is an independent execution of the reference procedure and is not independent evidence of the controller specification.

The three numerical paths exercise different parts of the contract. The resident path is incremental and uses streaming softmax on views. The training pass packs token nodes and gathers visible K/V into its local attention input. The auditor reinterprets the packed plan under the behavior model. Comparing the packed and resident outputs controls for a missing borrow or wrong predecessor view. Comparing the two training pullbacks in the fixed witness controls for a missing gradient route. Seeded replay checks sampled choices and logged rewards.

9.3 Observed results

Table 2 gives the counts and maximum errors for the two records. Both passed seeded regeneration under the recorded CPU profile. Direct replay of the sealed groups reproduced the next model and optimizer tensors exactly in both cases. The private-path column counts sampled occurrences outside the union of the final private ancestries of all branches. These actions remain owned

even when the final tails omit their nodes. In the combined witness, a terminal-private-path-only collector would miss eight of 69 sampled occurrences.

Quantity	Step 0	Step 1
Audited prompt groups	2	2
Regenerated episodes	8	8
Owned sampled actions	35	34
Actions outside final private paths	2	6
Mount occurrences across packed nodes	36	30
Rewind events	1	3
Maximum absolute behavior log-probability error	1.19×10^{-7}	2.38×10^{-7}

Table 2: Audit results for one CPU rank. Mount occurrences count entries in the plan's per-node mount lists. One logical read can contribute visible occurrences to multiple nodes.

The fixed double-precision witness gave a maximum packed-to-resident logit difference of 5.55×10^{-16} and a maximum packed-to-streaming parameter gradient difference of 2.78×10^{-16}. The reader's loss produced an isolated gradient norm of approximately 0.02923 at a producer-only token embedding. Blocking the mount while holding the same tokens and positions fixed made that gradient zero. The borrowed view shared the same backing allocation as the producer slice. These outcomes jointly exercise storage aliasing, exact graph membership, and the live numerical derivative.

We stopped a local run after the first completed step, resumed from its checkpoint, and compared the result with a two-step uninterrupted run. The model and optimizer tensors, next-step index, and stored provenance matched. The companion includes both post-update checkpoints, the two trace records, their audit outputs, and the evidence of this restart comparison. Cross-device tolerance and multi-rank behavior were outside the completed verification. A two-rank CPU attempt could launch processes, but local socket restrictions blocked collective initialization.

9.4 Mutation tests and claim boundaries

We altered one fact at a time in a valid step-zero record. Table 3 reports the observed detection stages. The tests demonstrate why the audit levels have separate output fields. They are targeted compiler and audit tests, rather than a measurement of detection power against arbitrary adversaries.

Mutation	Observed result
Decrease one finite behavior log-probability by 0.01	Structural replay accepted the legal value. Numerical replay rejected it. A raw-file digest anchored to the original record also rejected the change.
Flip the sign of one reward	Numerical replay accepted the unchanged logits and support. Seeded regeneration rejected the reward.
Point a read at an unpublished node	The Lean transition compiler rejected the record.
Repeat a JSON object key	The auditor rejected the input at parse time.

Table 3: Observed audit outcomes for four deliberately altered copies of a sealed record.

A useful additional negative control is a backend that detaches mounted K/V while returning the same forward arrays. It can pass a value-only audit. The producer-specific gradient witness is designed for that failure mode. A service that changes only the external retrieval result can preserve the policy's historical log-probabilities if the record is adjusted consistently. Detecting that case requires authenticated tool provenance or regeneration under the declared environment. Different negative controls isolate different checking layers.

9.5 Scaling the evaluation to a GPU cluster

A production validation starts with a frozen set of recorded episodes. The first pass compares a reference evaluator and optimized target at identical weights, dtype, tokenizer and model adapter. It checks action probabilities, representative cotangents by edge role, and the complete set of owned action identifiers. The second pass reruns branch controls with keyed random draws and sealed tool observations. The third exercises checkpoint and reducer failures under independent processes. Keep these passes separate from search-accuracy or cost experiments.

Report work by stage. Parsing and compilation use CPU time. Resident policy inference uses one or more devices. Packed replay and reverse accumulation consume the accelerator's memory and compute. Audit overhead includes storing manifests, sampling retained closures, and any re-execution. Record the number of bytes of each class and the amount of compute spent per audited episode or update. A small exact audit is a useful baseline for sampling checks on much larger runs, but those sampling checks have different statistical guarantees.

The deployment references a pinned Lean toolchain, a Torch installation profile, and a Kubernetes GPU Job [16, 17, 18]. The GPU manifest assumes a suitable device plugin, an image registry, and an exclusive output volume for the run. Its data-parallel configuration holds independent complete episodes on each rank. A graph-partitioned trainer would need the per-incidence transport and commit protocol in Section 6 and the cut witnesses of Section 5. These are specified as a design with proof obligations rather than claimed as deployed infrastructure.

10 Conclusion

A training update can be audited at several resolutions. An accepted trace supplies the controller's causal state and the unique ownership ledger. A translated plan supplies the numerical dependencies and the smooth functions evaluated at each gate. Behavior weights and a sampler profile permit numerical replay and seeded regeneration. A committed version and complete reduction manifest connect that evidence to an optimizer state. The formal results identify the conditions under which these witnesses compose and survive retries. The CPU artifact demonstrates the distinctions with a live borrowed gradient, deliberately altered records, and a restart from a durable checkpoint.

Authorship and artifact relationship. Prepared for anonymous review. The responsible author or research group is to be confirmed in the editorial submission record. An AI assistant contributed to writing, implementation, and artifact preparation. The companion compiler and reference training task are shared with the separate mathematical manuscript. This manuscript's original focus is replay certification, audit validation, and recoverable update publication.

A A complete audit witness

The fixed witness provides a simple trace with one rewind and a transitive mount. Root owns a two-token prompt and forks a scout and reader. The scout publishes four forced tokens at positions $2, 3, 4, 5$. The reader owns a task token at position 2. It then takes a detour containing one sampled action. After rewind its private cursor points back to the task and its next position is 3. The scout slice with local range $[1, 2)$ carries the native key at position 3. The reader's read therefore moves its next position to 4 while its private checkpoint is still the state after its own task.

The compiler inserts no extra token automatically. The trace supplies a forced bridge at position 4. The reader's sampled report uses the bridge output. Root waits for the scout and reader, then requests the report together with its previously borrowed slice. Its new view contains two mount occurrences and its own private ancestry. A forced synthesis token produces the logits for the sampled answer. The detour remains in the chronological plan and the masked score includes it. This witnesses a rewind that changes the active state without changing the probability of what happened before it.

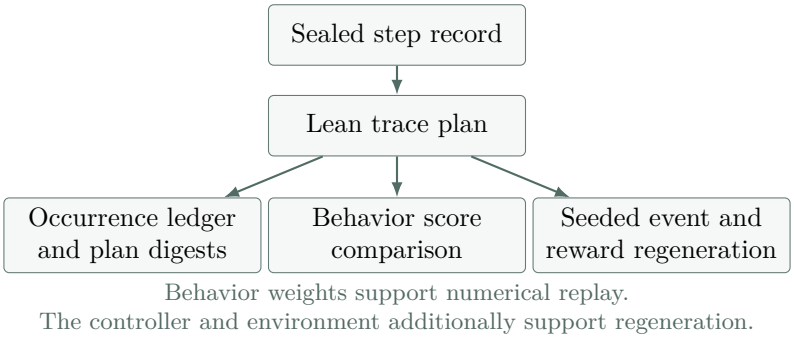


Figure 1: The auditor produces separately interpretable structural, numerical, and generative results. A separately trusted digest or signed statement can authenticate its inputs.

The packed evaluator uses one row for each owned token. Local attention prefixes may refer to rows of the same producer more than once, but the action ledger still contains one entry for each sampled occurrence. On this witness, the occurrence list has one detour action, one reader action, and one root answer. The scout token selected for the mount is forced and does not add to that count. Its parameter contribution through the reader comes entirely from the live borrowed data edge.

A malformed trace can demonstrate each operational check. Changing the source name to a future publication makes the read fail before allocation. Replacing the forced bridge by an immediately sampled first token makes the compiler reject its predecessor view. Restoring a scout checkpoint on the reader branch makes the private-history check fail. Giving an action repeated entries in its support changes its normalization and fails the legal support check. These are intentionally local errors whose diagnostic messages identify the event.

B Running the reference checks

The source companion has one directory for BranchCloud and one for the manuscript. Within the source directory, build the compiler and check the proof module with the pinned toolchain. Install the CPU wheel for the supplied replay evidence.

```
lake build
lake build BranchCloud.Certificates
python -m venv .venv
. .venv/bin/activate
python -m pip install torch==2.7.1 \
  --index-url https://download.pytorch.org/whl/cpu
python verify.py
```

The first audit uses the initial behavior model. The second uses the previous checkpoint. The evidence files and their results are included in the archive. A local rebuild may produce a different compiler binary fingerprint. The opt-in rebuild flag permits that comparison and preserves a separate result field indicating whether the executed compiler matches the recorded binary.

```
python audit_replay.py evidence/step-00000-rank-0.json \
  --initialize --regenerate --allow-rebuild \
  --verify-transition evidence/behavior-step-00001.pt
python audit_replay.py evidence/step-00001-rank-0.json \
  --behavior evidence/behavior-step-00001.pt \
  --regenerate --allow-rebuild \
  --verify-transition evidence/after-step-00001.pt
python audit_mutations.py evidence/step-00000-rank-0.json \
  --allow-rebuild
```

The numerical mode requires the recorded Torch version and device kind and matches the runtime source fingerprint. A user can instead create new evidence on their own configuration with the training command in the companion README. That record can be replayed with the exact local compiler and its own saved behavior checkpoint. The current implementation uses action seeds keyed by logical step and rank, so local repeatability assumes the same world size and trainer configuration. The cloud manifest is a launch template for a target cluster, whose GPU and networking behavior must be checked there.

C Implementation and proof boundary

Component	Evidence in the companion	Further conditions for a full system
Smooth kernel lowering	Shape checker, executable register plan and ordinary correctness argument	Realized primitive and derivative contracts for the chosen tensor backend.
Causal trace compiler	Executable transition validator, arithmetic and layout proofs, invalid trace witness	Capability policy and parser isolation for multi-tenant service use.
Resident and packed evaluators	Measured CPU agreement, alias identity and live producer gradient	Optimized model kernels, actual rotary profile and complete private state per architecture.
Replay certificate	Raw and ordered-plan digests, seeded regeneration and exact CPU update replay	A verified origin for expected digests and independent authentication for external effects.
Training recovery	Observed single-writer CPU restart and a container/job template	Target-cluster validation, fencing, and a linearizable version head for concurrent publication.
Cross-host gradient transport	Per-incidence manifest and proof of retry-safe complete reduction	Concrete transport, authenticated workers, and an implemented receiver ledger.

For reproducibility of the mechanical checks, the compiler uses Lean's standard library and needs no external math library. The proof module's ordered nonoverlap theorem concerns the defined contiguous interval layout. Its bottom-right theorem concerns a rectangular causal mask with a fully visible earlier prefix. Its append theorem concerns the stated pure partial transition interpreter. These lemmas are intentionally independent of Torch. The general translation criterion, distributed cut proof, and durable publication theorem are mathematical arguments whose conclusions have their respective explicit premises.

References

[1] Tianming Sha, Junnan Li, and Siqi Zhao. *Octopus: One Brain, Many Branches for Latent-KV Multi-Agent Reinforcement Learning.* Technical report, 2026. Full report and byline. Landing record and abstract. The landing record lists Tianming Sha.

[2] Amir Pnueli, Michael Siegel, and Eli Singerman. *Translation Validation.* Tools and Algorithms for the Construction and Analysis of Systems, 151–166, 1998. Publisher record.

[3] Xavier Leroy. *Formal Verification of a Realistic Compiler.* Communications of the ACM 52(7), 107–115, 2009. Author's paper.

[4] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM 21(7), 558–565, 1978. Author's paper.

[5] Songlin Yang, Jan Kautz, and Ali Hatamizadeh. *Gated Delta Networks: Improving Mamba2 with Delta Rule.* arXiv:2412.06464, 2024.

[6] Tri Dao and Albert Gu. *Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality.* arXiv:2405.21060, 2024.

[7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness.* arXiv:2205.14135, 2022.

[8] Luc Moreau and Paolo Missier, editors. *PROV-DM: The PROV Data Model.* W3C Recommendation, 2013. W3C specification.

[9] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. *in-toto: Providing Farm-to-Table Guarantees for Bits and Bytes.* 28th USENIX Security Symposium, 2019. USENIX publication.

[10] Anders Rundgren, Bret Jordan, and Samuel Erdtman. *JSON Canonicalization Scheme (JCS).* RFC 8785, 2020. RFC Editor.

[11] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. *Parallel Random Numbers: As Easy as 1, 2, 3.* SC11, 2011. Author project and publication link.

[12] Ben Laurie, Eran Messeri, and Rob Stradling. *Certificate Transparency Version 2.0.* RFC 9162, 2021. RFC Editor.

[13] K. Mani Chandy and Leslie Lamport. *Distributed Snapshots: Determining Global States of Distributed Systems.* ACM Transactions on Computer Systems 3(1), 63–75, 1985. Authors' paper.

[14] Maurice P. Herlihy and Jeannette M. Wing. *Linearizability: A Correctness Condition for Concurrent Objects.* ACM Transactions on Programming Languages and Systems 12(3), 463–492, 1990. Authors' paper.

[15] Paris Carbone et al. *Lightweight Asynchronous Snapshots for Distributed Dataflows.* arXiv:1506.08603, 2015.

[16] Lean contributors. The Lean Language Reference. Language documentation.

[17] PyTorch contributors. Torch distributed launcher and published 2.7.1 installation builds. Launcher documentation. Release installation matrix.

[18] Kubernetes contributors. Schedule GPUs. Kubernetes documentation.