

Observation-Preserving Runtime Gates and Checkpoint Semantics for Adaptive Learning

Version 1.1 - 12 October 2026 - Prepared for editorial review

Author: Anonymous

Abstract

We give a set-valued semantics for an adaptive learning runtime with optimizer memory, model revisions, request queues, and rejected events. Numerical data are opaque payloads with a separate validity-receipt interface. Projection to a discrete controller preserves one-step outcomes and finite transcripts. We prove typed checkpoint roundtrips, encoding injectivity, continuation preservation, and a gate that checks proposed transitions against the model. Lean 4.19.0 checks 21 new theorems; the same step and gate definitions are compiled into the live controller. Evidence includes 45 tests, 124,064 bounded controller comparisons, 300 generated events, and two NumPy/JAX scenarios. Compiler, JSON, Python adaptation, and numerical kernels remain explicit trust boundaries. The contribution is a checked instantiation combining control observations and complete checkpoint payloads in a correction-aware artifact.

1. Setting and prior work

The companion manuscripts separate higher temporal composition, typed tensor responses, and stateful objective queries [1-3]. This note connects a small learning runtime to an exact control model and a live correspondence gate in sets. General infinity-categorical coherence, real primitive derivatives, and uniform numerical simulation remain separate obligations.

Compiler semantic preservation and translation validation provide established approaches to implementation correspondence [4,5]. Here individual proposed transitions are checked against an executable model. The contribution is the checked instantiation with revision-sensitive requests, preserved optimizer payloads, and rejection observations; neither inductive refinement nor equality-based validation is claimed as new. Lean supplies proofs and controller source [6]; JAX supplies a separately tested numerical backend [7].

2. Full state, projection, and observations

Fix a program identity p and a finite list B of backend names. A ticket is $(i, r) \in \mathbb{N}^2$: identifier and revision. Pending requests and completions carry tickets and feature or output capsules. Capsules are finite strings. The adapter uses canonical JSON for float64 arrays, including shape and dtype; Lean leaves their numerical meaning uninterpreted.

Definition 2.1 (Full state). A state $s \in S$ contains program and backend identities; parameter, velocity, learning-rate, momentum, and query-history capsules; revision r ; capacity c ; next identifier n ; and ordered pending and completed lists. These retain every runtime checkpoint field, including objective, point, projection, response, and loss data in the query capsule. Retention does not prove numerical truth.

Definition 2.2 (Control projection). The map $\alpha : S \rightarrow C$ forgets capsules and retains program, backend, revision, capacity, next identifier, and ordered ticket lists. Distinct parameters or memories can

project equally, so control equality does not imply training equivalence.

The Boolean predicate V on C checks program p , membership of the backend in B , capacity at most 1024, pending length at most capacity, identifiers below n , distinct identifiers across both lists, pending revisions equal to r , and completion revisions at most r . Natural-number types supply nonnegativity. The full model's proved invariant is $V(\alpha(s))$. Shape, finiteness, magnitude, and hyperparameter conditions are additional Python-side numerical-domain checks, not predicates proved by this Lean invariant.

Definition 2.3 (Outcome). An outcome is (s', a, ρ) , where a is a Boolean acceptance flag and ρ is one of

$$\{\text{accepted, capacity, pending, stale, domain, unsupported}\}.$$

Its erasure is $\alpha_O(s', a, \rho) = (\alpha(s'), a, \rho)$. The law covers these normalized observations, not every Python diagnostic string or value; gradient-norm diagnostics are not certified.

3. Events and guarded transitions

Events contain their kind, payload capsules, and a Boolean receipt for supported numerical operations. Receipts represent proposal-domain checks, not derivative or rounding proofs.

- **Submit.** Discrete precondition: Pending length is below capacity; Candidate changes: Prepend (n, r) and its feature capsule; increment n .
- **Complete i .** Discrete precondition: i is pending; Candidate changes: Remove that pending ticket; append (i, r) and the output capsule.
- **Train.** Discrete precondition: Pending list is empty; Candidate changes: Replace parameter, velocity, and query capsules; increment r .
- **Reconfigure.** Discrete precondition: No additional event precondition; Candidate changes: Replace backend, capacity, rate, and momentum fields.

Domain-failure and unsupported events never commit a candidate. Reconfiguration retains parameters, velocity, queries, pending and completed requests, revision, next identifier, and program identity. Training does not cross a live request revision: it is blocked while any request is pending.

Write $K : S \times E \rightarrow S$ for full candidate construction, $\bar{K} : C \times \bar{E} \rightarrow C$ for control candidate construction, and $\epsilon : E \rightarrow \bar{E}$ for event erasure. Let $P(c, e)$ be the displayed discrete precondition and $N(e)$ the receipt. The acceptance test is

$$A(c, e) = P(c, e) \wedge N(e) \wedge V(\bar{K}(c, e)).$$

The full step $F(s, e)$ returns $(K(s, e), \text{true}, \text{accepted})$ when $A(\alpha(s), \epsilon(e))$ holds. Otherwise it returns s unchanged, with false acceptance and the appropriate failure tag. The controller step G has the same test and tags, using $\bar{K}$. A failed discrete precondition takes precedence over a bad numerical receipt. Thus invalid training data submitted while requests are pending still has the pending observation. Numerical-domain and configuration failures share the domain tag.

Proposition 3.1 (Frames and rejection). Rejected steps retain the full state. Every step preserves $V \circ \alpha$ from a valid initial state. Reconfiguration preserves the fields stated above, including velocity memory; training with a nonempty pending list is rejected.

Proof. Rejection selects the old state; acceptance requires V of the candidate. Reconfiguration assigns only its four fields. Training's precondition is false when requests are pending. Lean checks `rejected_unchanged`, `control_preserves`, `step_preserves`, `reconfigure_frame`, and `training_blocked`. $\square$

4. Observation-aware correspondence

Lemma 4.1 (Candidate erasure). For every state and typed event,

$$\alpha(K(s, e)) = \bar{K}(\alpha(s), \epsilon(e)).$$

Proof. Submit maps the new request to its ticket. Complete filters by identifier, so ticket mapping commutes with filtering and appending. Training changes only revision after erasure; reconfiguration changes backend and capacity. Failure constructors are identities. Lean checks the list-filter lemma and `candidate_refines`. $\square$

Theorem 4.2 (One-step correspondence). For all s, e ,

$$\alpha_O(F(s, e)) = G(\alpha(s), \epsilon(e)).$$

Proof. Both steps evaluate the same acceptance test. Lemma 4.1 identifies accepted successor controls; the other observations agree by definition. Rejection retains the old control and matching failure tag. This is `step_refines`. Structural equality needs neither initial validity nor receipt correctness; invariant preservation and the live adapter require a valid start. $\square$

Let $T_F(s, w)$ be the ordered list of outcomes obtained by running the finite event word w , using each outcome's state as the next input. Define T_G similarly for the controller.

Theorem 4.3 (Finite transcripts). For every finite w ,

$$\text{map}(\alpha_O, T_F(s, w)) = T_G(\alpha(s), \text{map}(\epsilon, w)).$$

If $V(\alpha(s))$ holds, every outcome state in $T_F(s, w)$ also satisfies the predicate.

Proof. Induct on the word. Theorem 4.2 identifies the first outcome and successor control; apply induction there. Proposition 3.1 supplies the successor invariant. Lean checks `transcript_refines` and `transcript_valid`, retaining rejected self-loops and their observations. $\square$

Corollary 4.4 (Path interpretation). Consider the directed transition graphs whose vertices are valid full or control states and whose edges retain the typed event and its normalized outcome. Erasure defines a functor between their free path categories. It preserves identities and finite path concatenation.

Proof. Theorem 4.2 gives an edge map with the correct endpoints, and Proposition 3.1 keeps the endpoints valid. Extend the edge map to words. Empty words map to empty words and mapping a concatenation is concatenating the mapped words. Theorem 4.3 identifies the corresponding observations. $\square$

This is a written categorical interpretation of the checked list laws, not a formalized Lean **Functor**, an equivalence, or a higher-coherence theorem. It supplies a set-valued fragment relevant to [1].

5. The checked runtime gate

Definition 5.1 (Gate). For a proposed full outcome o , define

$$\Gamma(s, e, o) = \begin{cases} \text{some}(o), & o = F(s, e), \\ \text{none}, & o \neq F(s, e). \end{cases}$$

Equality includes every full-state field, acceptance, and normalized reason. State equality uses the payload strings themselves, not only digests or the control projection. A matching rejected outcome is a successful gate check; a gate mismatch is an error that aborts the call.

Theorem 5.2 (Gate soundness and completeness). If $\Gamma(s, e, o) = \text{some}(u)$, then $u = F(s, e)$. The model outcome always passes. A differing proposal yields none.

Proof. The none branch cannot produce some. In the other branch, the equality test gives $o = F(s, e)$, while injectivity of some gives $u = o$. The remaining two statements are the true and false branches of the same test. Lean checks `gate_sound`, `gate_complete`, and `gate_mismatch`. $\square$

Consequently successful checks from a matched state and typed event stream give the model’s full outcomes and, by Theorem 4.3, its erased transcript. Applying this to native execution requires the trust boundary of Section 7; it is not static verification of unmonitored Python.

A velocity reset during reconfiguration can preserve every control field but fails the full-state frame check. Double revision increments and stale acceptance also fail. Training capsules enter through the proposal interface; the gate does not independently differentiate their objective.

6. Checkpoints and continuation

The typed wire algebra has string, natural-number, Boolean, and finite-array constructors. A state encoding is an array of 13 fields: a schema tag, seven strings, three natural coordinates, and the pending and completed arrays. Each request or completion is an array containing its identifier, revision, and payload string. The decoder requires the complete schema and exact constructor pattern. It rejects a missing velocity field or a Boolean where a natural coordinate is required.

Theorem 6.1 (Typed roundtrip and injectivity). For every full state s ,

$$\text{decode}(\text{encode}(s)) = \text{some}(s).$$

The state encoding is injective. Typed event encoding has the same roundtrip property.

Proof. Request and completion roundtrips reduce directly. List induction lifts a decoder’s left inverse to mapped lists; apply it to both state lists. Equal state encodings then decode to $\text{some}(s) = \text{some}(t)$, hence $s = t$. Event constructors reduce individually. Lean checks `decodeList_encode`, `state_roundtrip`, `encoding_injective`, and `event_roundtrip`. $\square$

Corollary 6.2 (Continuation). If decoding the encoding of s returns t , then $T_F(t, w) = T_F(s, w)$ for every finite typed event word w .

Proof. Theorem 6.1 identifies $t = s$; substitute in the transcript.

Lean: `decoded_checkpoint_continuation`. $\square$

Future typed events include their numerical proposals and receipts; independently rerun kernels need not supply identical proposals. The runtime test resumes the same backend and complete state. A parameter-only export discards memory and control fields and has no such continuation claim.

7. Executable bridge and trust boundary

The artifact contains a NumPy reference interpreter and a JAX lowering for the shared-parameter block

$$H = \tanh(XW + \mathbf{1}b^T), \quad Y = X + HW + \mathbf{1}b^T.$$

Momentum training uses $v' = \beta v + \nabla L(\theta)$ and $\theta' = \theta - \eta v'$ for a squared loss. The new `CheckedRuntime` first computes a Python proposal, normalizes a typed event and state, and sends both to the local controller. The controller invokes the actual compiled `checkedGate` definition. Python commits only after a matching response, full-state restoration check, and erased-observation check. Gate disagreement, unavailable service, or malformed response raises an error before the session updates its state or appends a normal transition record. There is no silent unmonitored fallback.

Accepted feature events normalize input lists into owned float64 arrays before capsule construction. Negative integer completion requests are always unknown; the adapter represents them by the unused next identifier for the same stale observation. Other malformed identifier types have a domain observation.

The raw attempted event remains in the Python audit log. The typed event codec theorem concerns typed events, not injectivity of this raw-input normalization.

The checked source imports bundled `Std` and requires no `Mathlib`. Lean 4.19.0 checks 21 new theorem declarations in `Correspondence.lean`; the unchanged first kernel supplies 14 earlier theorems. There are no proof holes or custom application axioms. Printed transitive dependencies include standard `propext` and `Quot.sound`. The compiled program uses the checked transition and gate definitions. The Lean compiler, C compiler/linker, native execution, JSON parser/printer, Python adapter, and numerical-domain receipt computation remain trusted. A build report binds the current source digests to the controller digest; this is an integrity record relative to trusted digests, not authentication or a compiler-correctness proof.

Concrete JSON and Python-to-wire roundtrips are tested, not proved, including 2^{80} identifiers, escaped payloads, malformed arities, and Boolean/natural distinctions. The runtime is single-threaded and synchronous; numerical kernels, resource exhaustion, concurrency, and external effects are outside the proofs.

8. Reproducible evidence

Python 3.12.14, NumPy 2.3.5, and CPU JAX/JAXlib 0.4.38 were used. Dependency versions, sources, controller hash, and outputs accompany the artifact. A fresh isolated installation of the same JAX pins resolved an earlier JIT process crash; its cause was not established and the failure is recorded. The local Lean build used a documented startup-only process-path adapter; the official kernel and libraries were unchanged. The native controller runs without that adapter. A deliberately false equality is rejected as a proof-checker negative control.

- **Combined suite.** Result and scope: 45 tests pass, with no skips; includes original numerical checks, checkpoint resume, and live gate tests.
- **Bounded model/adapter enumeration.** Result and scope: 4,184 valid controls and 124,064 transitions agree with an independent Python table; counters 0-3, revisions 0-2, capacities 0-3, proposed capacities 0-4; both registered backend labels; all pending/completed orderings and allowed completion revisions.
- **Generated live runtime.** Result and scope: 300 seeded events produce 300 matching gate checks; 80 accepted; endpoint valid, with every step validated.
- **Integrated trajectories.** Result and scope: Both NumPy-start and JAX-start runs have 42 matching checks, 38 accepted and four rejected, 31 updates, and 62 objective-query records.
- **Numerical comparison.** Result and scope: Maximum observed final parameter/velocity discrepancy 1.12×10^{-16} (rounded upward), within the declared 2×10^{-11} tolerance.
- **Defect injections.** Result and scope: Double revision, stale acceptance, and velocity reset are blocked; the interrupted session does not commit the proposed transition.

The enumeration includes both receipt values, supported event kinds, unknown completions, illegal capacity changes, absent backends, and explicit domain/unsupported events. Numerical payloads are symbolic in that enumeration; it is not an exhaustive test of real or floating-point arrays. These measured runs establish neither a uniform numerical refinement bound nor general optimizer convergence. ## 9. Correction-aware compatibility

The supplied source release preserves the original manuscripts and examples and records reruns of their bounded checks. This journal attachment retains the executable examples; historical manuscripts and the full earlier ZIP remain in the source release, outside this attachment. The ledger retains its five old records and adds C006 (exact natural ticket metadata), C007 (the stronger controller/codec/gate scope), and C008 (a witness to an unpublished raw-list/normalized-array adapter defect).

Earlier results retain their applicable hypotheses; invalid assertions retain witnesses and replacements, not a claim of truth. The old queue's unknown-completion self-loop does not specify the new stale observation. The ledger records research lineage, not equivalence of all old and new theories.

10. Claims and remaining obligations

The completed milestone proves exact erasure, discrete invariants, payload frames, gate soundness, and typed codec laws, and executes the gate with tested adapters. Static Python verification, verified JSON/extraction, real reverse correctness, uniform floating-point guard margins, and higher coherence remain separate obligations.

Automation disclosure. OpenAI ChatGPT/Codex assisted with implementation, proof development, manuscript drafting, reference checking, and artifact verification. The submitting owner authorized the Anonymous byline and submission. No independent referee report or acceptance decision is asserted.

References

1. Anonymous. [Higher Lenses and Temporal Descent for Certified Adaptive Systems](#). Higher Categorical Semantics (2026), editorially reviewed. The historical source release contains version 1.1.
2. Anonymous. LensIR: Compositional Reverse Semantics for a Typed Tensor Fragment. Unpublished companion manuscript, version 1.1 (2026); retained in the historical source release.
3. Anonymous. Oracle and Jet Semantics of Stateful Optimization. Unpublished companion manuscript, version 1.1 (2026); retained in the historical source release.
4. Xavier Leroy. *Formal verification of a realistic compiler*. Communications of the ACM 52(7), 107-115 (2009). [doi:10.1145/1538788.1538814](#).
5. Amir Pnueli, Michael Siegel, and Eli Singerman. *Translation validation*. TACAS 1998, LNCS 1384, 151-166. [Institutional record](#).
6. Leonardo de Moura and Sebastian Ullrich. *The Lean 4 Theorem Prover and Programming Language*. CADE 28, LNCS 12699, 625-635 (2021). [doi:10.1007/978-3-030-79876-5_37](#).
7. JAX developers. [Forward- and reverse-mode autodiff in JAX](#). Implementation versions are pinned in the artifact; the linked documentation is current.