Buffer/Compute/Flush

Higher-Categorical Feedback and Higher Monads for Differentiable Compilation

Anonymous author
Research manuscript, version 1.0.0 — November 3, 2026
Prepared for *Higher Categorical Semantics*

Abstract. We formulate buffer/compute/flush as a repeated interaction of signals, executable code, state, and output routes. A buffer acquires a typed signal window; a computation acts on that window and the retained state; a flush sends results to storage, to an observer, or into a subsequent buffer. The same cycle therefore describes a stream processor, a recurrent neural network, and the execution of a staged tensor program, provided their distinct observations are recorded. Code–data interaction is expressed by evaluation. Ordinary temporal convolution is recovered under linearity and time-invariance assumptions, rather than postulated for every program.

We give a layered semantics using an operational category of typed buffer interfaces, coherent algebra objects in endofunctors, indexed effects, and guarded feedback. A higher monad organizes substitution of buffered programs; an algebra or an effectful interpretation realizes a selected computation. Monad multiplication, output routing, device dispatch, completion, and durable commit consequently have different types. Higher cells record compatible reorganizations of computation, while directed refinements remain distinct from equivalences. Sheaf and higher-topos interpretations are described with explicit descent and smoothness hypotheses.

The differentiable fragment has tensor types, products, higher-order functions, smooth primitives, directional differentiation, and scalar reverse gradients. Its executable implementation specializes source closures to a tensor single-assignment plan, constructs derivative programs, and trains a recurrent network across twelve rebuffering cycles. The reference experiment uses 795 parameters and preserves differentiation through state routing. A NumPy runtime was exercised; an optional PyTorch CUDA runtime executes the same plan, with GPU dispatch and synchronization represented separately. No GPU measurement or machine-checked higher-categorical proof is claimed.

Scope and status. This is a semantic framework and a working restricted compiler, not a completed verification of a GPU toolchain. General preservation statements are given as conditional obligations; extended proofs and elementary finite illustrations are intentionally omitted. The executable example is the recurrent training system itself. Publication and editorial acceptance are not asserted.

Keywords: feedback; buffering; higher monads; guarded recursion; differentiable lambda calculus; automatic differentiation; tensor compilation; CUDA; recurrent neural networks; higher topoi.

2. Problem, contributions, and reading map

The central proposal is a cycle: signals are buffered, code operates on buffered data, and the resulting signals are flushed to a destination that can supply the next computation. A flush may therefore close an episode without terminating the overall process. A neural hidden state and a sensor window are both buffers in this sense, although their ownership and update rules can differ. The theory must explain repeated interaction, not merely delayed evaluation.

Three questions guide the manuscript. First, what structure composes such cycles while retaining the distinction between code, data, and state? Second, which changes of buffering and scheduling preserve a selected observation? Third, when can differentiation pass through a routing boundary so that a compiled feedback system can be trained? These questions require separate semantic layers: control flow, mathematical values, physical execution, and evidence about implementation.

The proposed contribution is their integration around a typed interface. There is no claim that monads, guarded recursion, automatic differentiation, or asynchronous GPU execution originate here. The higher-monadic account builds on established definitions [M91, L17]; the feedback discipline and differentiation semantics draw on independently developed foundations [B12, HSV20, V21]. The specific buffer/compute/flush organization and its restricted executable realization are the objects being proposed.

Pages	Purpose
3–8	Signals, phase transitions, routing, code interaction, convolution, feedback.
9–17	Higher interfaces, monads, execution algebras, indices, effects, coherence.
18–20	Sheaves, internal reasoning, and the compilation contract.
21–26	Differentiable source language and compatibility with flushing.
27–34	Recurrent training, compilation, GPU boundaries, resources, failures.
35–38	Executable artifact, measured evidence, limitations, authorship.
39–40	Verified bibliographic links and their role in the framework.

We use *higher* in a precise, variable-dependent sense. A monad on an ∞-category carries coherent associativity data; an operational refinement may instead live in a directed higher category. A finite Python graph does not thereby become an implementation of every such higher cell. Its role is to realize a meaningful fragment at the ordinary computational level.

All real-valued equations describe ideal mathematics unless a numerical interpretation is explicitly selected. All effects are ordered unless a commutation law is supplied. All infinite feedback claims require productivity or a justified fixed-point structure. These conventions prevent the notation from silently asserting termination, exact arithmetic, or durable execution.

The intended reading is compositional. A reader interested in the abstract model can follow the interface and algebra constructions independently of the backend. A reader reproducing the experiment can follow the source term, derivative graph, and epoch loop independently of the sheaf extension. The compilation contract states where those readings meet: the same typed payload and feedback dependence must survive each interpretation.

3. Signals, buffers, and the retained world

Fix types I of input signals, H of retained computational state, O of outputs, D of stored records, and C of code. A buffer is a typed organization of signals, not necessarily an unstructured list. Possible interfaces include bounded windows, batches, channels with timestamps, tensor views, and references to immutable device arrays. We write Buf_I for the chosen interface and keep its policy as part of the specification.

An ingress operation has a declared type

$$\beta : \mathrm{Buf}_I \times I \longrightarrow \mathrm{Buf}_I .$$

For a window, β appends a signal and removes an expired one. For a bounded queue it may reject overflow. If acquisition can fail, the codomain must include that effect. No equation here makes overflow disappear. The buffer can carry validity masks and logical timestamps; these belong to the data used by later operations, rather than to undocumented host behavior.

A world sufficient to determine one cycle can be represented as

$$Q = H \times \mathrm{Buf}_I \times C \times D \times V \times R,$$

where V contains version identifiers and R contains runtime resources or capabilities. This is a product presentation of an interface; a dependent presentation is needed when the shape of a buffer depends on its version. Physical allocation is not identified with a mathematical component of Q. A device pointer can realize a component without being its semantic value.

The retained state must include every variable on which future behavior depends. For stochastic code, that includes an explicit random state or an effect that accounts for fresh randomness. For adaptive training, it includes optimizer moments, parameter versions, and the data-order policy. A scalar loss alone is therefore not a sufficient state description for a training system. The feedback interface makes this insufficiency visible.

Semantic ownership. A buffer can be read repeatedly when its content is immutable. A mutable buffer requires an ownership or interference discipline. We distinguish sharing an immutable value from simultaneously granting two writers access to one location. The reference compiler uses immutable input snapshots and new parameter arrays, so source-level sharing does not authorize mutation.

An implementation map ρ relates admissible machine states to Q. It can forget allocation addresses while retaining shapes, versions, and logical contents. Different ρ define different observations. A model that forgets queue length can support a value-preservation argument, but cannot by itself support a bounded-memory claim. This observation discipline will remain essential when program transformations are introduced.

A signal envelope can be written (ι, t, v, a), with stream identity ι, logical timestamp t, version v, and payload a. An ingress policy must state whether envelopes are ordered by arrival or by timestamp. If late arrivals can revise a past window, the retained world must record the revision policy. The pure recurrent experiment uses an already ordered fixed sequence, so its ingress convention has no arrival-order ambiguity.

4. The three phases as a transition discipline

We distinguish phases B, C, and F for acquiring or preparing a buffer, computing, and routing the result. A configuration records a phase and enough state to continue:

$$(\mathsf{B}, q, i) \longrightarrow (\mathsf{C}, q_b) \longrightarrow (\mathsf{F}, q_c, o, r) \longrightarrow (\mathsf{B}, q', i').$$

The last input may arrive later. The diagram specifies causal order, not a requirement that every physical operation block the host. Multiple cycles can overlap only when their dependencies and resource policies permit it.

Let M be a chosen effect monad on a suitable semantic category. Ingress, computation, and flushing can then have types

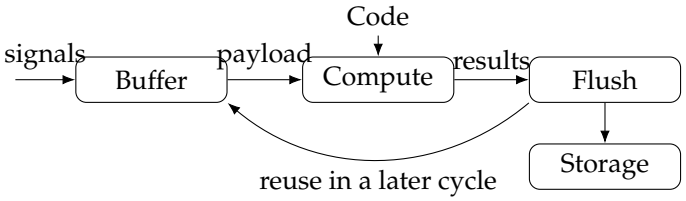
$$\begin{aligned} b &: Q \times I \longrightarrow MQ_b, \\ k &: Q_b \longrightarrow M(Q_c \times O \times \mathsf{Route}), \\ f &: Q_c \times O \times \mathsf{Route} \longrightarrow M(Q \times O). \end{aligned}$$

The phase interfaces Q_b, Q_c can enforce invariants that would be obscured by using Q everywhere. Kleisli composition gives the one-cycle map $\gamma = f \circ_M k \circ_M b$. The output copy retained in the codomain is a designated observation, not necessarily an additional physical copy.

A phase transition can emit a trace event with an epoch, route, and operation identifier. Such traces make buffer boundaries observable when required. They also explain why two programs with the same final tensor need not be equivalent as interactive systems. One might expose a partial output before a failure while the other exposes none. A value-only semantics deliberately forgets this difference and must say so.

Cancellation and failure require explicit transitions. If computation has only been enqueued, abandoning its host handle does not prove that its device work was cancelled. If flushing has reached an external store, an exception does not imply rollback. A phase diagram with no error edges is therefore a model of successful executions under assumptions, rather than a description of all runtime behavior.

We regard a complete successful cycle as one logical transition. It need not be an atomic transaction. Atomicity is an additional observational property of an implementation and its environment. This distinction permits the same abstract cycle to cover local immutable execution and distributed routing without attaching unjustified transactional guarantees to either.



5. Flush destinations and the meaning of reuse

A route specifies what becomes available after a computation. At minimum, we distinguish a route to a later computational buffer, a route to an observer, and a route to persistent storage. These destinations may be combined, but their effects must remain ordered when order is observable. Writing a tensor to an application queue and committing bytes to durable storage are different operations even if both are called flushing.

For a pure rebuffering route between compatible representations, let

$$r_{A,B} : A \longrightarrow \mathrm{Buf}_B, \qquad d_B : \mathrm{Buf}_B \longrightarrow B.$$

The required value-level equation is $d_B r_{A,B} = u$ for a declared conversion $u : A \to B$. For identity reuse, $u = \mathrm{id}_A$ after identifying the logical payload types. This equation does not force an implementation to copy the payload. A view or a reference can realize it if its lifetime and aliasing conditions are satisfied.

A storage route instead produces a receipt:

$$p : A \longrightarrow M\mathsf{Receipt}(A).$$

Its specification must define whether the receipt means accepted, written to a process buffer, acknowledged by a server, or committed under a durability contract. An observation operation can retrieve a value or evidence about its availability. The receipt is not simply a value of A, and its existence does not establish exactly-once effects.

Reusing a buffer preserves dataflow when the next computation receives the same logical value. It preserves differentiation only when the relevant dependence is also retained. Copying a numerical tensor normally preserves the mathematical identity map. Detaching it from a derivative graph changes the differentiable program even though the primal values agree. This distinction becomes decisive for recurrent training.

The reference runtime supplies logical rebuffer markers inside a tensor plan and a separate outer dispatch operation. Its markers preserve the identity map on tensors. It does not implement a persistent storage service, and it does not equate those markers with disk flushes. A successful repeated call on one cached dispatch returns the same result handles; it does not promise that a second process can recover them after a crash.

This treatment supports different operational realizations of one routing interface. Implementations may move bytes, transfer ownership, or retain an immutable reference. Their equivalence depends on the selected observation: payload equality alone, payload and lifetime, or payload and a storage receipt with stronger contractual meaning.

Serial routes obey a compatibility requirement. If a buffer conversion $u : A \to B$ is followed by $v : B \to C$, the combined payload route must realize $v \circ u$. A buffer label can remain visible even when u is the identity. This makes it possible to preserve the causal identity of a route while changing its physical representation, provided all observers use the declared interface.

6. Code–data interaction as an evaluation action

The phrase that code is convolved with data expresses a useful operational intuition: neither an inert code representation nor an unprocessed signal determines an output alone. We formalize their interaction by a typed evaluation action. In a closed semantic category, a pure instance is

$$\mathrm{ev}_{A,B} : [A, B] \times A \longrightarrow B.$$

An effectful code interface instead provides $\mathrm{ev}^M : \mathrm{Code}(A, B) \times A \to MB$. A decoder from code representations to $[A, MB]$ can determine this map. We do not assume that every mathematical function has a finite code or a computable decoder.

A buffered program can take both persistent state and a signal window:

$$\kappa : \mathrm{Code}(H \times \mathrm{Buf}_I, H \times O) \times H \times \mathrm{Buf}_I \longrightarrow M(H \times O).$$

The new state is subsequently routed to a future buffer, while O may be observed or stored. Parameterized neural code fits this interface through a map $\Theta \to \mathrm{Code}(H \times I, H \times O)$, provided the chosen representation supports the desired smooth interpretation in Θ.

There are two levels of code in the executable example. Source lambdas are finite syntax with lexical environments. Specializing their known function arguments produces a tensor plan. That plan then acts on numerical inputs. Substituting a function into a source combinator changes the program being generated; changing a weight tensor changes the data supplied to the generated program. These operations can be related without identifying their types.

An additional program can modify code itself. Its output must inhabit the same code interface or come with a typed translation to a new one. This makes adaptive code generation expressible, but does not establish that the change improves a loss or terminates. Differentiating a family of real parameters inside a fixed code structure is different from differentiating arbitrary discrete source rewrites.

The evaluation action preserves the intended meaning of code–data interaction while avoiding a false linearity assumption. A program may branch, normalize, or maintain state. Such behavior generally cannot be represented by a fixed linear temporal kernel. When a specific convolution algebra is available, evaluation can factor through its action on signals; the next section states those additional assumptions.

For reasoning about compilation, this action is the key square: interpreting the generated code and then supplying its data should agree, under the selected observation, with interpreting the original source application. The mathematical requirement is a compatibility of interpretations, not a claim that physical code and data have become the same object.

For composable pure code families, the interaction law is

$$\mathrm{ev}(c_2, \mathrm{ev}(c_1, a)) = \mathrm{ev}(c_2 \circ c_1, a).$$

For effectful code, the left side is Kleisli sequencing rather than an ordinary nested application. The code-composition operation must agree with that sequencing. This is precisely where a compiler that treats all inputs as inert data can lose the ordering of reads, writes, or parameter updates. The admitted pure tensor fragment gives a controlled setting in which ordinary composition is sufficient.

7. When interaction is literally convolution

Suppose signals take values in a finite-dimensional real vector space V, outputs in W, and code is a finite causal kernel $K_j \in \mathrm{Lin}(V, W)$ for $0 \le j < m$. A window buffer exposes $x_t, \dots, x_{t-m+1}$. Computation can then be specified by

$$y_t = (K * x)_t = \sum_{j=0}^{m-1} K_j x_{t-j}.$$

The boundary convention for negative indices is part of the interface. Zero padding, a supplied prehistory, and periodic extension define different systems. Flushing can store y_t and update the input window for time $t+1$. This is a literal convolution realization of the three-phase cycle.

Linearity and time invariance explain the kernel presentation. Without these hypotheses, a family κ_t can still act on buffered signals, but it need not commute with translation or preserve addition. A nonlinear recurrent network is therefore covered by the evaluation model of page 6 even when it is not a convolution operator. A convolutional neural layer has literal convolution or cross-correlation structure before its nonlinear activation; the complete network usually does not.

Composing finite linear causal kernels yields a kernel with ordered matrix products:

$$(L * K)_n = \sum_{i+j=n} L_i K_j.$$

The summands are typed $V \to U$ when $K_j : V \to W$ and $L_i : W \to U$. Associativity follows from the specified algebraic operations and finite index rearrangement. Matrix-valued kernels need not commute. The causal support records which buffered signals can influence an output.

There is a broader kernel calculus when the ambient enrichment admits the required coends. For profunctor kernels, composition has the schematic form

$$(L \odot K)(a, c) = \int^{b} K(a, b) \otimes L(b, c).$$

This is a separate construction, with its own variance and cocompleteness conditions. It does not turn every executable program into an analytic convolution. A measure-theoretic integral requires still more assumptions; the coend notation alone supplies no probability measure.

This distinction is useful for compiler design. A literal convolution can be lowered to a specialized accelerator primitive. General code–data interaction can instead be lowered to an appropriate graph of operations. Both preserve a buffer interface, but their optimization laws differ. Interchanging nonlinear operations with a sum, or exchanging the order of matrix kernels, requires a separate justification.

The manuscript consequently uses *interaction* for the general action and *convolution* for an explicitly specified kernel algebra. This retains the motivating intuition while making its mathematically stronger instances identifiable and testable.

For a finite kernel, differentiating with respect to its coefficients gives $\dot{y}_t = \sum_j \dot{K}_j x_{t-j}$; differentiating the signals gives $\dot{y}_t = \sum_j K_j \dot{x}_{t-j}$. These are concrete instances of bilinear code–data interaction. The recurrent example replaces this fixed bilinear action by a smooth state-dependent evaluation.

8. Feedback, causality, and guarded iteration

Once a computation flushes its state into a later buffer, the system has a feedback loop. In a deterministic pure presentation,

$$(h_{t+1}, o_t) = F_c(h_t, i_t).$$

The delayed use of h_{t+1} is essential. The equation computes a future state from available data; it does not ask the present state to solve an arbitrary instantaneous fixed-point equation. A fixed initial h_0 and input prefix determine every finite prefix of this recurrence.

For infinite streams, one needs a semantic account of productivity. A guarded setting supplies a later operation $\triangleright$ and permits feedback only through a delayed state channel. Schematically a feedback component has type

$$A \times \triangleright H \longrightarrow B \times H.$$

The associated guarded fixed-point structure must be supplied by the model. The topos-of-trees approach is an established source of such structure [B12]. An arbitrary higher topos is not asserted to have the same guarded fixed-point operation.

A trace operation on a suitable monoidal category offers another abstract description of feedback [JSV96]. Its axioms describe the compatibility of feedback with composition and parallel structure. They do not license a total trace on the cartesian category of all smooth maps: an arbitrary smooth instantaneous feedback equation need not have a solution. We therefore separate an assumed trace structure from the concrete causal recurrence.

The executable fragment uses a statically bounded unfolding. Twelve applications of a recurrent source function yield a finite acyclic tensor plan. Each new hidden state is routed through a logical rebuffer marker. This executes feedback over a finite horizon without requiring a runtime solver for recursive equations. It is nontrivial recurrent training, but it is not an implementation of productive infinite streams.

The distinction matters for gradients. Differentiating a finite unfolding uses a finite chain rule. Passing to an infinite horizon would require a notion of convergence and conditions justifying limits of derivatives. Contractivity, stability, and integrability are possible hypotheses, not consequences of the word feedback. The higher-monadic syntax of finite program substitution does not remove those analytic obligations.

A code update can itself close an outer feedback loop: an optimizer observes loss-derived information, modifies parameters, and supplies the next epoch. This outer loop is separate from the recurrent state loop. Their clocks, state spaces, and derivative boundaries must remain explicit.

At the stream level, causality can be expressed by prefix compatibility. If two input streams agree through time t, their output prefixes through time t agree, for the same initial state and code. A transformation of buffering is causal only if it respects that condition. A batch implementation may read a complete sequence already in memory, but the mathematical recurrence can still obey prefix compatibility because later input leaves do not influence earlier hidden-state nodes.

9. A category of typed buffer interfaces

An interface specifies input payloads, output payloads, retained state, routes, and an observation policy. A morphism is a buffer/compute/flush component respecting those specifications. Serial composition connects a declared output route to an admissible downstream buffer. Parallel composition juxtaposes components whose resources are independently usable. If two components share a mutable location, an additional interference discipline is needed before parallel composition is defined.

Ignoring effects for the moment, stateful components can be represented by

$$f : A \times H_f \longrightarrow B \times H_f, \qquad g : B \times H_g \longrightarrow C \times H_g.$$

Their serial composite acts on $A \times H_f \times H_g$ by running f and then g, retaining both state components. Reassociation of the state product provides the familiar structural isomorphisms. An effectful version uses a strong monad and preserves the indicated execution order.

These presentations are not identified merely because their state spaces have the same dimension. To identify components, choose an observational relation: equal output streams from related initial states, equal traces, or a simulation compatible with a world interpretation. Hidden-state representations can differ while external behavior agrees. Conversely, equal immediate outputs can coexist with different future behavior.

The buffer boundaries are part of the interface when latency, traces, or ownership transfer are observed. A compiler may erase a logical boundary only after specifying which observations forget it. In the reference implementation, boundaries are identity nodes with labels, so numerical observations may forget them while a route trace can still record them. This makes the relation between a diagram and an executable graph concrete.

A higher presentation replaces a mere set of equivalent implementations by a space of implementations and compatibility witnesses. A chosen equivalence can then carry a record of how shapes, routes, and observations align. However, an ordinary category of components can always be used as a restricted starting point. Passing to its nerve supplies a legitimate ∞-category, with no assertion that additional nontrivial higher information has been constructed.

Thus the categorical perspective first organizes compositional interfaces. The higher perspective is warranted when families of equivalences and their compatibilities are actually supplied. It is not warranted solely by drawing more levels of arrows around an unexamined runtime.

A useful invariant of serial composition is an interface relation $R_f \subseteq H_f \times \widehat{H}_f$ connecting two representations. If each component carries related states to related states and agrees on the routed output, the product relation $R_f \times R_g$ is the candidate relation for the composite. Any failure or observation effects must be included in that compatibility. This identifies the exact data a compositional simulation argument would use, instead of requiring the two implementations to have identical internal buffers.

10. Higher cells and the observations they preserve

For a chosen higher semantic category $\mathcal{C}$, the mapping space $\mathrm{Map}_{\mathcal{C}}(A, B)$ organizes components from A to B. Points are morphisms; paths are equivalences between these morphisms in the mapping space; higher paths compare such witnesses. A numerical implementation can be mapped into this structure only after its interpretation and observations are fixed. Different physical schedules do not automatically determine paths.

Suppose $\mathsf{Impl}(A, B)$ is a space of admissible implementations and $\mathsf{Obs}(A, B)$ a space of observed behaviors. A fiber over behavior u,

$$\mathsf{Real}(u) = \mathsf{Impl}(A, B) \times_{\mathsf{Obs}(A,B)} \{u\},$$

retains both an implementation and evidence that it realizes u. In a genuinely higher setting this is a homotopy pullback. The fiber can be empty, disconnected, or contain nontrivial higher identifications. Its contractibility must not be assumed merely because all members pass one experiment.

Useful observations include exact real denotation, a full interaction trace, and a numerical error specification. These are not interchangeable. An error tolerance usually defines an estimate or relation, not a transitive equality: two errors of size ε can accumulate to 2ε. Approximate equivalence therefore needs a quantitative composition rule rather than automatic treatment as an identity type.

Directed program improvements require particular care. A transformation that removes an unnecessary allocation might be irreversible as an operational refinement, although its source and target denote the same pure function. If such transformations are represented as noninvertible 2-cells, an $(\infty, 2)$-categorical setting or a suitable enriched directed structure is needed. In an $(\infty, 1)$-category, cells above dimension one are invertible. We do not encode an arbitrary refinement as an invertible homotopy simply to use higher notation.

For buffer/compute/flush, a coherence witness can compare two ways of flattening nested code or composing serial routes. It must respect the chosen observation and all dependent interfaces. A further witness can compare competing coherence witnesses. This describes the role of higher data in a compiler contract; constructing that data is a separate task.

The executable artifact records graphs, versions, and route labels, but no machine-readable space of such coherence witnesses. Its tests provide evidence about particular interpretations. They do not populate every higher fiber or establish a complete higher-categorical adequacy theorem.

The observation choice can be organized by successive forgetful maps: full execution records map to routed payload histories, which map to terminal values when those values exist. Equality at a richer level implies equality after applying a forgetful map. The converse requires additional information and generally fails. This hierarchy explains why a compiler can preserve a final loss while changing intermediate availability. A higher equivalence over the terminal-value observation should therefore not be reused as a witness over the full execution record without an additional lifting argument.

11. The higher monad of buffered programs

An explicit sequential higher monad is available before adding lambda binding. Let K be a space of command descriptions and put $K^* = \coprod_{n\geq 0} K^n$. On spaces define

$$\mathbb{T}_K X = K^* \times X, \qquad \eta(x) = ([], x), \quad \mu(u, (v, x)) = (u +\!\!+ v, x).$$

Ordered concatenation gives the multiplication and its coherent associativity. A non-discrete K retains families and paths of commands. This is the free unary-command higher monad: words list operations from outermost to innermost syntax. It is not a universal model of branching or lambda binding.

A higher monad, including $\mathbb{T}_K$, is an associative algebra object

$$T \in \mathrm{Alg}_{E_1}(\mathrm{End}(\mathcal{E})),$$

where the monoidal product is composition [L17]. Its unit and multiplication have types

$$\eta : \mathrm{id}_{\mathcal{E}} \Rightarrow T, \qquad \mu : T \circ T \Rightarrow T.$$

Associativity and unitality come with coherent higher compatibility. Merely writing the ordinary monad equations in a homotopy category would discard some of this data.

For a fuller free syntax interpretation, choose a presentable ∞-category $\mathcal{E}$ of typed families and an accessible signature P whose free monad T exists [PP03]. Then TX is a family of finite programs with placeholders in X. The map η_X inserts a placeholder as a trivial program. The map μ_X substitutes programs for placeholders, flattening a program whose open leaves contain further programs. Neither operation has the type of executing a neural tensor kernel. In particular, flattening a code tree does not consume a GPU stream or produce a durable storage receipt.

The signature can expose buffered reads, pure operations, and routes. Phase information can be included as sorts. Constructors with bound variables require a binding-aware signature, for instance on a context-indexed presheaf category. A polynomial endofunctor on one untyped set is not a complete treatment of typed lambda binding.

The free-monad presentation organizes finite syntax. Feedback is introduced through a state channel and a guarded or otherwise justified iteration structure; it is not obtained simply by allowing infinitely nested applications of μ. The finite recurrent program in the artifact is consistent with this distinction: its source function is unfolded to a finite graph before runtime execution.

Raw syntax may be discrete. Additional higher program equalities require an enriched presentation, localization, or semantic target carrying them. An extension must specify its equations and their soundness. The explicit $\mathbb{T}_K$ construction supplies one genuine higher-monadic instance; the full binding-aware construction remains a separate obligation.

12. Flush as an algebra and execution as interpretation

For a T-algebra with carrier A, an evaluation map has type

$$\alpha : TA \longrightarrow A, \qquad \alpha\eta_A \simeq \mathrm{id}_A, \quad \alpha\mu_A \simeq \alpha T\alpha.$$

The coherence is part of the algebra structure. In this interpretation, the carrier already supports the operations needed to realize the buffered program. The second equation says that interpreting nested syntax in stages agrees with interpreting its substitution-flattened form.

For the concrete $\mathbb{T}_K$, supply an action $K \to \mathrm{Map}(A, A)$. Extend it to words by $\alpha([k_1, \ldots, k_n], a) = k_1(\cdots k_n(a))$. Buffer, compute, and route commands act on a phase-tagged world; invalid phases can enter an explicit error summand. The empty word acts identically and concatenation acts by ordered composition. Thus a chronological buffer–compute–flush cycle is the syntax word $[\mathsf{flush}, \mathsf{compute}, \mathsf{buffer}]$. Its innermost operation runs first. This constructs an execution algebra for that sequential fragment.

This is a useful formal account of one meaning of flushing: a buffered description is realized in a chosen carrier. It does not give a natural operation $TX \to X$ for every arbitrary X. A carrier without the required operations has no reason to admit that map. Nor does it mean that all physical flush destinations share one algebra structure.

Effectful execution can instead be expressed by a monad morphism

$$\tau : T \Rightarrow M.$$

For a term $p \in TX$ and environment $\xi : X \to MA$, its interpretation is

$$[\![p]\!]_\xi = \mu_A^M\big(M\xi(\tau_X(p))\big) \in MA.$$

The formula is typed: $\tau_X(p)$ lies in MX, $M\xi$ sends it to MMA, and the runtime multiplication sequences the result. Interpreting free generators in M is subject to the chosen equations and coherences.

The routing operations of page 5 occur inside the interpreted computation. A term that requests storage is realized by the storage handler. A term that reuses a tensor is realized by the appropriate buffer route. Their shared syntax organization does not make their operational guarantees identical. A single carrier can support several handlers, but their results and observations still need explicit types.

For the executable fragment, the source-to-plan compiler and plan executor provide ordinary interpretations of a restricted syntax. We do not claim to have constructed a higher-monad morphism for every binding constructor and every runtime effect. The algebraic account identifies the laws that a future verified interpretation would have to satisfy.

This discipline rules out a common collapse of meanings: μ flattens syntax, α evaluates in a carrier, a routing handler moves results, and an await operation establishes completion. One implementation can expose them through a convenient interface, but a preservation argument must keep their distinct domains and codomains visible.

13. State, traces, exceptions, and a runtime monad

A concrete runtime needs more than pure value functions. As an ordinary set-based illustration of its shape, let S be runtime state, $\mathcal{L}$ a trace monoid, and E a set of errors. Consider

$$MX = S \longrightarrow \big((X + E) \times S \times \mathcal{L}\big).$$

Its return operation leaves S unchanged and emits the empty trace. Its bind operation runs the first computation, concatenates traces in order, and continues only on a successful result. On error, it retains the state and trace already produced. This describes one error policy, not an axiomatic requirement for every runtime.

Retaining state on failure is deliberate. If an operation has submitted work or emitted a record before reporting an error, a semantics that silently discards all state would misdescribe its environment. A rollback policy can be represented separately when a real transactional mechanism supports it. The combination of state and exceptions must therefore be selected with the operational contract in mind.

An asynchronous value in X may be a handle rather than a completed tensor. The state S then includes queues and dependency events. Dispatch returns a handle, whereas awaiting it transitions to a state in which a completion condition has been established, or produces an error. This separation fits the CUDA execution model without pretending that a returned host object proves device completion.

For higher semantics, the ingredients must be interpreted in a category where the relevant products, coproducts, exponentials, and monoidal trace structure exist. Higher coherence then refines the ordinary bind laws. The displayed formula is not asserted to construct the intended asynchronous runtime monad in every ∞-category. It gives a transparent reference contract against which such a construction can be assessed.

The order of trace concatenation matters. A storage acknowledgement followed by an observation cannot usually be exchanged with the opposite order. Commuting independent pure tensor nodes is a narrower transformation than commuting arbitrary runtime effects. A strong monad permits parameterized sequencing, but strength does not by itself imply commutativity.

The artifact implements a smaller state machine around a pure plan: buffered, enqueued or completed, and failed. A failed dispatch object is not automatically retried. That rule does not establish atomic failure; it simply prevents an ambiguous partial execution from being silently submitted again through the same object.

The Kleisli composite has the explicit form $g \circ_M f = \mu^M \circ Mg \circ f$. When f acquires a buffer and g computes from it, the intermediate value is a valid phase interface, not a global state silently shared by every component. The runtime state carried by M is an additional layer. This distinction permits a pure signal state to be replayed while a runtime trace continues to record that replay as a new execution.

For a deterministic single-stream runtime, a handle can be modeled by an identifier together with a dependency node in S. Awaiting the handle must refer to that node's completion condition, not merely to the existence of its identifier. A future implementation of this abstract runtime could replace device-wide synchronization by a specific event without changing the completed-value interpretation.

14. Nested buffers, substitution, and retained sharing

Nested buffering arises when a generator constructs an intermediate program, when several windows form one batch, or when a pipeline component returns code for a later cycle. These situations need not have the same type. A buffer of signals, a syntax object with buffered leaves, and a runtime handle for a future result must remain distinguishable even when the implementation uses similar containers.

For syntax substitution, the governing diagram is

$$TTTX \underset{\mu_{TX}}{\overset{T\mu_X}{\rightrightarrows}} TTX \xrightarrow{\mu_X} TX.$$

The two composites are coherently equivalent by the monad structure. The diagram compares two orders of flattening nested program placeholders. It does not authorize running the contents in an arbitrary order. An effectful interpretation must map this coherence to the corresponding sequencing coherence in the runtime.

Data buffering can have its own aggregation operation, such as concatenating windows with compatible timestamps. That operation may fail on a shape mismatch or change a batching policy. Calling both aggregation and syntax substitution multiplication does not prove that they commute. A bridge between them must specify how a data batch is reflected in the source interface and in the generated program.

Sharing is another independent issue. A syntax tree can duplicate a placeholder, while a compiled graph can reuse one already computed tensor. For pure deterministic terms with immutable inputs, reuse can preserve value denotation. If the placeholder denotes a random draw, a mutable read, or a storage action, duplication and reuse can have different observations. Consequently, graph sharing is justified for a pure fragment or by a specified effect analysis.

The reference compiler retains source let bindings as shared graph references, but it does not globally merge all common subexpressions. Its training plan contains one forward loss evaluation and another forward evaluation used by the reverse transform. This is a simple implementation choice, not evidence that duplicated forward work is semantically necessary. A future elimination pass would need to preserve parameters and epoch snapshots.

This layered interpretation makes nested buffering useful without treating it as a universal effect eraser. Code flattening controls substitution, buffer aggregation controls data organization, and graph sharing controls reuse of computations. Higher coherence can relate well-specified combinations of these operations only after their individual laws are available.

One can make graph sharing explicit with a context of named nodes. A let extension adds a typed node to that context; a later use is a reference to the extension rather than a re-execution. Forgetting names maps the presentation toward a tree unfolding, but that unfolding can duplicate substantial work. A compiler contract can preserve value semantics under this forgetful map while retaining a separate resource semantics for the graph. This explains why substitutional equality and an allocation or work bound should not be conflated.

15. Indexed and graded organization of cycles

A single endofunctor T suppresses information that is important in a feedback runtime: the phase before and after a computation, the parameter version it reads, and the resources it consumes. An indexed family can retain such information. Let $\mathcal{I}$ be a category of admissible interface states, and suppose suitable indexed computation objects $T_{i,j}$ are given with composition maps

$$T_{j,k} \circ T_{i,j} \longrightarrow T_{i,k}, \qquad \mathrm{id} \longrightarrow T_{i,i}.$$

The maps and their coherence encode which transitions can be sequenced. This is additional structure, not automatically obtained by renaming one ordinary monad.

An index can contain a phase and an epoch. A program reading parameters at version v can return a gradient explicitly tagged with v. An update operation expecting version v can then reject a gradient from a different version. The interface rules prevent a stale gradient from being mistaken for a current one; they do not prove that a distributed implementation enforces the rule unless version checking is actually implemented.

Grades describe another dimension of information. For a monoidal grade structure $(G, \otimes, e)$, a graded monad has units and multiplication maps of the form

$$\eta : \mathrm{id} \Rightarrow T_e, \qquad T_g T_h \Rightarrow T_{g \otimes h}.$$

Possible grades include a sequence of effect labels or an upper bound on logical work. A resource bound must define how grades combine under sharing, parallel execution, and failure. A numerical label attached to a graph is not by itself a verified resource analysis.

For recurrent computation, a horizon grade can record how many signal cycles are unfolded. Serial composition adds horizons only when the interfaces agree about the state passed between them. Parallel composition often combines horizons by a different rule. Thus a grade adequate for one composition operation need not describe the other.

The artifact has ordinary metadata for epochs and route labels, together with fixed tensor shapes. It does not implement a dependent type system of runtime capabilities or a complete graded higher-monad library. The indexed presentation specifies how that extension could give buffer and flush boundaries a more precise static meaning.

Keeping these indices visible also clarifies training: recurrent state versions evolve inside a parameter epoch, while optimizer steps create new parameter versions. Treating both clocks as one unqualified iteration counter would erase information needed for correct data reuse and gradient interpretation.

For example, an index may be (v, t, ϕ), with parameter version v, recurrent time t, and phase ϕ. A hidden-state route advances (v, t, F) to $(v, t+1, \mathsf{B})$, while an optimizer update advances (v, T, F) to $(v+1, 0, \mathsf{B})$. These transitions encode two distinct clocks. They also determine which saved hidden states are admissible for replay: a state from another parameter version generally denotes a different trajectory, even when its shape matches.

16. Combining staging and effects requires a law

Let T organize buffered syntax and let M organize runtime effects on the same category. Their composite MT does not automatically inherit the intended monad structure. A distributive law, with a fixed orientation, can supply the necessary interaction:

$$\lambda : TM \Rightarrow MT.$$

It must respect both units and both multiplications, together with higher coherence in the higher setting. Intuitively it explains how an effectful placeholder inside syntax is moved into an effectful program representation. This is stronger than having unrelated maps μ^T and μ^M.

For example, the two unit requirements have ordinary shadows

$$\lambda \circ \eta^T M = M\eta^T, \qquad \lambda \circ T\eta^M = \eta^M T.$$

The multiplication requirements compare distributing a nested layer at once with distributing it in stages. These equations govern a proposed interaction; the manuscript does not assert that an appropriate λ exists for arbitrary buffer operations and arbitrary effects.

The obstacle is operational. If a source term uses a placeholder twice, moving an effectful read outside the term can change two reads into one. With mutable state or randomness, the observed outputs can change. Similarly, staging a storage action and executing it while staging are different behaviors. A purported universal distribution law would erase distinctions the runtime is supposed to preserve.

There are several legitimate restricted choices. One can require placeholders to be pure immutable values. One can use a linear discipline for selected effectful inputs. One can keep staging and execution separate and interpret syntax directly into M, as on page 12. Or one can construct a combined effect theory with the desired interaction equations. The appropriate choice depends on the application and is part of its semantics.

The reference compiler takes the first and third routes: its numerical source fragment is pure, input bindings are snapshotted, and an external dispatch object runs the generated plan. It therefore does not need to commute arbitrary host effects through source lambda abstraction. This restriction is a concrete reason that higher-order specialization is manageable.

For a more general feedback compiler, the necessary law must include the interaction of parameter versions, routing, and failure. It must explain whether a flush can be moved across an update and what observations are preserved. No associativity equation for code substitution can substitute for these runtime-specific obligations.

The direct interpretation alternative has explicit ordinary coherence equations. A monad morphism $\tau : T \Rightarrow M$ satisfies

$$\tau_X \eta^T_X = \eta^M_X, \qquad \tau_X \mu^T_X = \mu^M_X \, M\tau_X \, \tau_{TX}.$$

The second equation starts at TTX on both sides. It compares substituting syntax before interpretation with interpreting its layers and sequencing the effects. In the higher setting the equations are equipped with compatible witnesses. Their types expose which interpretation is being composed and exclude an untyped appeal to flattening.

17. Coherent transformations and directed refinements

Compiler transformations compare programs at several levels. Reassociation of a serial pipeline may express a structural equivalence. Reordering independent pure tensor nodes may preserve the value of a graph while changing its operational schedule. Removing a logical route marker may preserve tensors but change a trace. A lower-precision kernel can change the value while satisfying an explicit error bound. One undifferentiated notion of equality cannot express all these situations.

Fix an observation functor O from a category of program presentations to a semantic category. A proposed equivalence $u : p \simeq q$ needs a compatible witness $O(p) \simeq O(q)$. For two transformations u, v and their composites, a coherence witness must compare the resulting semantic identifications. The data are especially important when transformations change parenthesization or pass through nested buffering layers.

Associativity provides a characteristic coherence problem. Four components admit several reassociation paths. A chosen associator must satisfy the pentagon, and higher refinements must satisfy the corresponding coherent extension. An E_1-algebra organizes this hierarchy for monadic composition. It does not establish that an independently coded optimization pass respects it.

Directed transformations can instead be judged by a relation $p \preceq q$, such as a simulation or a resource refinement at equal denotation. If operational refinements are noninvertible 2-cells, their composition belongs in a directed higher framework. The value interpretation may send them to equalities even though the operational presentation does not identify them. This separates a semantic quotient from a record of how an implementation was improved.

For buffer/compute/flush, useful transformation obligations include preserving a state route, retaining a version dependency, and not moving a storage effect before the computation that justifies it. A route-preserving certificate must refer to actual route identifiers and compatible payload types. A statement that two graphs have the same number of buffers supplies none of that information.

The restricted artifact performs source closure specialization and derivative expansion. It retains a topologically ordered tensor graph but does not implement an optimizer with higher coherence certificates. The mathematical account specifies how such certificates could be organized; it is not a report that they have already been generated.

This distinction also limits the phrase verified compiler. Passing integration checks supports particular behaviors. A fully verified implementation would additionally connect source typing, every lowering rule, route handling, and runtime assumptions through machine-checkable preservation evidence.

A transformation that preserves a complete route history can be composed with one that preserves only its payload projection, but the resulting guarantee is the weaker payload guarantee. Thus the observation interface is itself part of a transformation's type. This gives a practical rule for assembling optimization passes: a later pass cannot recover a stronger guarantee merely because an earlier pass supplied it.

One can retain certificates in a category over the observation interfaces. A certificate includes its source and target programs, the observation map, and the comparison witness. Composition uses the explicitly compatible observation maps. The higher extension then compares compositions of certificates at this same interface, rather than forgetting their domains during coherence reasoning.

18. Sheaves of buffers and contextual execution

Buffer contents and code capabilities can depend on context. A tensor may be available only on a device that owns its allocation; an output route may be available only where a storage capability has been granted. A site $(\mathcal{J}, J)$ can encode a chosen notion of compatible local contexts. We may then work in

$$\mathcal{E} = \mathrm{Sh}_\infty(\mathcal{J}, J).$$

This is a modeling choice [L09] whose covering families must be specified. Network connectivity alone does not define a Grothendieck topology.

A buffer sheaf B assigns local buffer data to each context and restriction maps to context changes. The descent condition says that compatible local data glue, with the higher coherences required by the chosen sheaf theory. It does not state that arbitrary physically distributed buffers agree. Agreement is part of the descent datum, and existence of a global section can fail when the local requirements conflict.

For code and buffers to interact internally, their operations must respect restriction. An internal evaluation map, for example,

$$\mathrm{Code}(A, B) \times A \longrightarrow MB,$$

must be defined in $\mathcal{E}$, with an internal runtime effect structure M. Defining an unrelated runtime separately at every context does not ensure that the resulting family is a sheaf or that its operations glue. Compatibility on overlaps is a substantive requirement.

Local compilation data can likewise glue only under hypotheses. Local plans must agree along overlaps, their interface translations must be compatible, and the interpretation must respect the chosen descent. Even then, the global object obtained is a semantic object. Turning it into one deployable executable can require additional realizability and resource conditions.

A change of context represented by a geometric morphism supplies a left-exact inverse-image functor. It preserves finite-limit structure, but does not automatically preserve every exponential, chosen real object, or runtime monad used in this manuscript. A base-change theorem for the compiler therefore needs explicit preservation hypotheses. Merely placing the construction in a higher topos does not make it invariant under every change of environment.

This perspective is useful when a feedback system is assembled from context-dependent interfaces. It records which local signals and code can be combined, and what compatibility is required. It does not supply a distributed consensus protocol or make absent capabilities locally available. The executable example remains a single-process realization; the sheaf account is a conditional extension of its interface semantics.

For a cover $\mathcal{U} = \{U_i \to U\}$ with the required fiber products, write $\check{C}(\mathcal{U})$ for its Cech nerve. Descent has the form

$$B(U) \simeq \lim_{[n]\in\Delta} B\big(\check{C}(\mathcal{U})_n\big),$$

where evaluation on a coproduct means the product of the corresponding local evaluations. A compatible family includes overlap identifications and their higher compatibility. When the cover represents device contexts, those identifications must relate actual admissible payloads and capabilities. This equation specifies gluing of compatible data; it gives no operation for manufacturing missing data.

19. Internal logic, time, and smooth structure

The internal language of a topos can express predicates about buffers: shape compatibility, availability of a route, ownership hypotheses, and relations between versions. Such predicates are interpreted in context. For example, a condition $\mathsf{Ready}(b, v) \Rightarrow \mathsf{Readable}(b)$ is meaningful only after readiness and readability have been defined for the chosen world. Its internal validity does not identify readiness with physical completion.

A natural number object can index finite histories and parameter epochs. It supports the internal formulation of bounded iteration and arithmetic constraints. Additional structure is needed to model productive streams, fair scheduling, or an actual hardware clock. In a guarded model, a later modality can record that a state becomes available in the next logical step. A storage availability modality would be a different construction with different laws.

Smooth interpretation also requires more than the ambient topos axioms. The object called $\mathbb{R}$ must be selected, together with a notion of smooth map and a derivative structure. One may use sheaves on suitable smooth test spaces, or the diffeological semantics used for higher-order differentiation [HSV20]. These choices have their own embeddings, mapping objects, and tangent behavior. They are not all interchangeable.

In particular, the existence of an internal real ring does not imply the Kock–Lawvere axioms or provide a nontrivial object of infinitesimal dual numbers. Conversely, the finite-dimensional tensor derivative used by the artifact does not require synthetic differential geometry. Its semantics can be given by ordinary smooth real maps after specialization. The broader geometric setting is useful for organizing higher-order function spaces.

Classical arguments also require care in an internal setting. An internal existential statement need not supply an externally computable global witness. A proposition that every local buffer has some admissible route does not by itself yield one uniform executable route selector. Extraction requires a constructive representation and a realizability or compilation argument appropriate to the source logic.

The proposed compiler therefore has two explicitly separated obligations: logical terms must be represented by the admitted source fragment, and that fragment must be interpreted by the runtime. No claim is made that arbitrary proofs in an arbitrary higher topos can be converted into GPU programs. The paper's executable contribution is narrower and concrete: a typed smooth tensor fragment with statically specialized higher-order functions and finite recurrent training.

An internal invariant can be stated as a subobject $\mathsf{Valid} \hookrightarrow Q$. Preservation by a cycle means that the cycle, restricted to valid worlds and admitted inputs, factors through the chosen object of valid successor computations. For a pure cycle this is an ordinary factorization through $\mathsf{Valid} \times O$. For effects, the notion of valid computation must account for error branches and observations. A predicate about successful results alone does not rule out invalid intermediate external writes.

The factorization formulation is useful because it identifies the term that must be constructed to establish preservation. It also shows what a compiler must transport: a witness of validity in the source world must correspond to an appropriate witness after runtime realization.

20. The semantic compilation contract

Let $\mathsf{Src}(\Gamma, A)$ contain well-typed source terms and let $\mathsf{Plan}(\Gamma, A)$ contain admissible tensor plans. A compiler $\mathcal{K}$ maps the selected source fragment into plans. Source and target interpretations should fit a compatible comparison

$$[\![e]\!]_{\mathrm{src}} \simeq [\![\mathcal{K}(e)]\!]_{\mathrm{plan}}.$$

The comparison is indexed by environments satisfying source shapes and by the observation being preserved. Real denotation, route traces, and floating-point results require different versions of the contract.

Conditional preservation statement. Assume every primitive interpretation has the stated type and denotation; closure specialization respects lexical binding and evaluation order; product and let lowering preserve sharing where effects are observable; the plan scheduler respects all data and effect dependencies; and the runtime realizes the selected operations. Under a compositional interpretation satisfying these obligations, compilation preserves the selected source behavior. This statement specifies a proof target. A complete general proof is not supplied here.

For differentiation, add a derivative interpretation of each smooth primitive and a compatibility condition for the derivative transformation. For routing, add the equation identifying a pure rebuffer with its declared value map and derivative. For asynchronous execution, distinguish the denotation of an enqueued handle from a completed value. These extra premises are necessary: value preservation of an ordinary forward graph alone says nothing about a gradient or a storage receipt.

In the higher formulation, these comparisons should be natural in admitted contexts and compatible with substitution. Transformations of terms and their coherence witnesses should map to corresponding semantic witnesses. This is stronger than proving equality of one closed output. The restricted code does not implement the full higher comparison; it exposes the interfaces a future formal development would need.

The artifact provides practical evidence for source shape checking, specialization, graph execution, forward directional differentiation, reverse gradients, and state-route dependence on one recurrent model. Its generated plan is a finite trusted local artifact. The JSON reader checks ordering but is not a hardened validator for adversarial programs. No execution of arbitrary source strings is used.

The contract also identifies the numerical boundary. The mathematical comparison concerns ideal real primitives. A numerical runtime replaces these by floating-point implementations. Relating the two requires a separate error analysis with bounds on primitive errors and their propagation through feedback. Passing the reported numerical checks is useful evidence; it is not a universal floating-point correctness proof.

At the ordinary machine boundary, let ρ interpret an admissible machine world and let $\mathsf{step}_{\mathrm{mach}}$ execute one completed cycle. The corresponding simulation target is

$$\rho_{Q\times O}\,\mathsf{step}_{\mathrm{mach}} = \gamma\,\rho_{Q\times I}$$

on the specified domain, with effects interpreted on both sides. For asynchronous execution this diagram applies after the declared completion observation; a separate diagram relates enqueued handles. Separating these diagrams prevents a submission event from being used as a witness of a completed source transition.

21. A differentiable lambda-calculus fragment

The executable language is a smooth, simply typed lambda-calculus fragment with explicit differentiation forms. It is not a complete implementation of the algebraic differential lambda calculus of Ehrhard and Regnier [ER03]. The relationship is the use of typed functions together with a directional differential operation. Our practical fragment has a finite-dimensional tensor interpretation and a restricted staging discipline suitable for neural training.

Types are generated by

$$A, B ::= \mathsf{T}_s \mid (A_1 \times \cdots \times A_n) \mid A \to B,$$

where s is a fixed scalar, vector, or matrix shape with positive dimensions. An empty shape denotes a scalar. Write Euc for types built from tensors and finite products, excluding arrows. Their interpretations have a canonical product vector-space structure. Function types can occur in source programs but do not cross the final numerical runtime boundary.

Terms include variables, lambda abstraction, application, let binding, product formation and projection, and smooth tensor primitives. They also include a labelled route $\mathsf{rebuffer}_\ell(e)$, a directional form $\mathsf{jvp}(f, a, v)$, and a scalar gradient form $\mathsf{grad}(f, a)$. The main derivative typing rules are

$$\frac{f : A \to B \quad a : A \quad v : A \quad A, B \in \mathsf{Euc}}{\mathsf{jvp}(f, a, v) : B \times B}, \qquad \frac{f : A \to \mathsf{T}_() \quad a : A \quad A \in \mathsf{Euc}}{\mathsf{grad}(f, a) : A}.$$

The gradient uses the chosen Euclidean inner product to identify a cotangent with a tensor of the same shape.

The primitive vocabulary contains equal-shape addition, subtraction, and pointwise multiplication; matrix multiplication; vector bias addition to a matrix; hyperbolic tangent; and mean reduction. Bias broadcasting is explicit in the source typing rule. Arbitrary implicit broadcasting, data-dependent shapes, division, nonsmooth branching, and unbounded recursion are absent from the executable fragment.

Admission also requires no nested source differentiation while specializing an AD function body. A type can express more terms than this prototype admits; the compiler enforces the additional staging restriction. This is an implementation boundary, not an assertion that higher derivatives are mathematically undefined.

These constructs already express higher-order neural combinators, parameter products, nonlinear recurrence, a scalar training objective, directional sensitivity, and full parameter gradients. The executable therefore exercises actual differentiable program compilation rather than illustrating buffering with a scalar arithmetic toy.

For instance, the matrix primitive has the static rule

$$\frac{a : \mathsf{T}_{(m,n)} \quad b : \mathsf{T}_{(n,p)}}{\mathsf{matmul}(a, b) : \mathsf{T}_{(m,p)}}.$$

A Euclidean parameter product has dimension $N_A = \sum_i \prod_{d \in s_i} d$, counting the scalar empty product as one. The source gradient returns that product structure instead of flattening it into one untyped vector. This preserves the interfaces of the five neural parameter blocks and permits shape-specific updates while retaining the same mathematical Euclidean pairing.

22. Higher-order specialization and lexical meaning

The source language evaluates function structure during staging. A lambda is represented by its body and lexical environment; application extends that environment with the actual argument. Tensor values at this stage are symbolic graph references. A higher-order combinator can therefore receive source functions and construct a graph without receiving any numerical tensor contents.

For compatible tensor types X, H, the recurrent example uses

$$\mathsf{compose} = \lambda f : H \to H.\lambda g : X \to H.\lambda x : X.\, f(g(x)).$$

Each cycle supplies a nonlinear activation as f and a state-dependent affine function as g. The state and weight references are captured by the affine function's lexical environment. The resulting function is applied to the buffered signal for that cycle.

Let binding controls sharing. A source subterm assigned to a hidden-state variable is lowered once in that evaluation and subsequent uses receive its graph reference. Replacing this mechanism by textual substitution of unrestricted effectful computations could duplicate effects. Our source fragment is pure at the tensor level, and a rebuffer marker records a logical route rather than performing an external write during staging.

Freshness matters when differentiation interacts with captures. Suppose $f = \lambda p.F(p, q)$ and the actual point is the same graph reference as the captured q. The derivative with respect to p must hold q constant. Identifying the formal parameter with the captured reference would incorrectly differentiate both uses. The compiler inserts fresh identity aliases for the formal tensor leaves before specializing the body of an AD form.

For reverse differentiation, these aliases are the requested derivative boundary. For forward differentiation, they receive the supplied tangent seeds; captured inputs receive zero tangent unless explicitly included in the formal parameter. This implements a partial derivative of the function as expressed, even when source environments share numerical values. It also makes the role of an AD boundary independent of a physical tensor copy.

The source fragment has no general recursive function generator. A Python construction function assembles twelve typed source applications and let bindings before compilation. The numerical runtime then receives an ordinary first-order graph. The higher-order capability demonstrated here is genuine source-level function passing and lexical specialization, not execution of arbitrary function-valued tensors on the GPU.

The specialization boundary is checked at the entry output: its type must be Euclidean. A function can be constructed, passed to another source function, and applied during staging, but a remaining function value is rejected as a runtime output. Input environments likewise contain only Euclidean leaves. These conditions distinguish specialization from a general closure-conversion compiler that would carry code pointers and environments into the tensor runtime.

The ordinary evaluator uses call-by-value ordering for applications: the function position is lowered before the argument, and the resulting closure is then applied. This ordering makes the prototype's source interpretation definite even though its admitted numerical terms are pure.

23. Smooth denotation and the higher-order boundary

For each tensor shape s, interpret T_s as the corresponding finite-dimensional real vector space. Products receive product spaces. The tensor primitives denote their usual smooth maps. A fully higher-order interpretation needs a category that supports function spaces as well as a suitable notion of smoothness; finite-dimensional manifolds alone are not cartesian closed.

Diffeological semantics provides one established approach to this issue [HSV20]. The role of that reference is to support a principled higher-order semantic setting, not to transfer an unproved correctness theorem to this new implementation. Once a closed source function has been specialized to the admitted tensor graph, its mathematical denotation is a finite composition of smooth finite-dimensional maps. This narrower fact is sufficient to state its ordinary directional derivative and gradient.

For a first-order function $f : A \to B$ between Euclidean types, the directional interpretation is

$$[\![\mathsf{jvp}(f, a, v)]\!] = \big([\![f]\!](a), D[\![f]\!](a)[v]\big).$$

For scalar output and the selected Euclidean pairing,

$$\langle [\![\mathsf{grad}(f, a)]\!], v\rangle = D[\![f]\!](a)[v].$$

This identification depends on the inner product. A coordinate-free cotangent has a dual type; returning the same tensor type is justified here by the explicitly fixed Euclidean representation.

The route $\mathsf{rebuffer}_\ell$ denotes the identity on its payload for the pure training interpretation. Its label is observed in a separate route trace. The smooth payload interpretation can forget that trace. A storage route would generally not be a smooth real-valued function and is not admitted inside this pure differentiation fragment.

Higher smooth structure and higher program equivalence are separate questions. A smooth path of parameter values is a family in a chosen geometric setting. A compiler correctness witness is an identification between interpretations. One does not automatically supply the other. An enriched semantic framework may relate them, but its enrichment and coherence must be specified.

The artifact thus has a clear mathematical boundary: it compiles specializable higher-order source terms into finite smooth tensor programs, then differentiates those programs. It does not implement arbitrary function-space derivatives, distribution-valued derivatives, or synthetic infinitesimal reasoning inside a higher topos.

On the specialized finite-dimensional fragment, tangent transport can be packaged as

$$\mathsf{T}f(x, v) = (f(x), Df(x)[v]).$$

The chain rule states $\mathsf{T}(g \circ f) = \mathsf{T}g \circ \mathsf{T}f$, and $\mathsf{T}\,\mathrm{id} = \mathrm{id}$. Products are transported componentwise. This gives a precise compositional meaning to forwarding primal and tangent buffers together. The notation T here denotes the tangent construction and is distinct from the program monad T. Their interaction requires the comparison described on page 26, rather than an identification of the two functors.

24. Forward differentiation as a staged program

For tensor leaves x with tangent $\dot{x}$, forward transformation assigns each graph node a primal and tangent. The central rules are ordinary chain-rule identities written as graph constructors:

$$\begin{aligned} D(AB)[\dot{A}, \dot{B}] &= \dot{A}B + A\dot{B}, \\ D(a \odot b)[\dot{a}, \dot{b}] &= \dot{a} \odot b + a \odot \dot{b}, \\ D\tanh(a)[\dot{a}] &= (1 - \tanh(a)^2) \odot \dot{a}, \\ D\,\mathrm{mean}(a)[\dot{a}] &= \mathrm{mean}(\dot{a}). \end{aligned}$$

Addition, transpose, broadcasting, and reduction are transformed by their linear actions. All shape rules remain explicit; the tangent of a matrix has the same shape as its primal matrix in the selected representation.

The implementation does not approximate these derivatives by finite differences. It constructs a new tensor graph containing the derivative operations. A later runtime executes that graph with either NumPy arrays or PyTorch tensors. Central differences appear only in an independent numerical check of one compiled recurrent objective.

A source jvp first introduces fresh formal aliases, specializes the function body at them, and propagates seeds from those aliases. Unseeded external inputs are constant with respect to that partial derivative. Constants inserted by the compiler have zero tangent. This process differentiates a function of the selected parameter product while holding the signal sequence, target, and initial state fixed.

For a pure route,

$$D\mathsf{rebuffer}_\ell(a)[\dot{a}] = \mathsf{rebuffer}_\ell^T(a, \dot{a})_{\mathrm{tangent}} = \dot{a}$$

at the payload level. A richer implementation can route primal and tangent buffers together with their corresponding metadata. The prototype preserves the payload dependency and does not create a separate tangent routing trace at every marker.

The formal forward transform is linear in the tangent argument for a fixed primal point. The tensor implementation may show floating-point rounding error in checks of this linearity. Exact linearity belongs to the real denotation; numerical tolerances belong to the backend contract. The distinction is especially relevant when successive feedback cycles amplify small errors.

Because the program is staged, tangent propagation can be combined with later graph optimization. Such an optimizer would need to preserve both the primal and derivative denotations. The supplied compiler favors a transparent graph over fusion: it does not claim specialized native CUDA code generation or a performance advantage over an established autodifferentiation framework.

For the neural parameter product, a direction is a tuple $(\dot{W}_x, \dot{W}_h, \dot{W}_o, \dot{b}_h, \dot{b}_o)$. Shared use of a weight at twelve cycles shares its tangent seed as well. The resulting directional derivative includes the effect of changing that same parameter at every use, rather than changing one time-local copy. The compiled JVP therefore checks both the derivative rules and the treatment of parameter sharing across feedback routes. Its scalar output is contracted from the complete finite-horizon dependence, not just the final affine layer.

25. Reverse differentiation and accumulated cotangents

For a scalar objective, reverse transformation propagates a cotangent from the output toward the inputs. A tensor reused at several downstream nodes receives the sum of their contributions. This accumulation is essential for recurrent weights, which are shared across all time steps. Replacing the sum by an overwrite would train a different program.

For $Y = AB$ with cotangent $\bar{Y}$, the Euclidean rules are

$$\bar{A} \mathrel{+}= \bar{Y}B^{\mathsf{T}}, \qquad \bar{B} \mathrel{+}= A^{\mathsf{T}}\bar{Y}.$$

For $Y = \tanh(A)$, $\bar{A} \mathrel{+}= \bar{Y} \odot (1 - Y^2)$. For a matrix bias broadcast $Y_{ij} = A_{ij} + b_j$, the bias cotangent sums along the batch axis. For a mean of n entries, the input cotangent is the broadcast output cotangent divided by n.

These rules produce another finite tensor plan, rather than delegating gradient construction to framework autograd. The reverse pass visits reachable primal nodes in reverse topological order and accumulates graph references for their cotangents. It stops propagation through the fresh formal aliases selected as derivative arguments. The runtime later executes the constructed operations as ordinary tensor work.

Reverse derivative categories provide an abstract setting for reverse operations [C20]; higher-order reverse AD has additional semantic requirements [V21]. Our artifact uses higher-order specialization before first-order reverse transformation. It does not claim a general implementation of the full higher-type reverse translation in those references.

At a rebuffer identity, the cotangent is routed back unchanged. A detached route would instead block that contribution, altering the objective's dependence on earlier states. The artifact includes a check that its recurrent weight gradient is nonzero and a stronger directional agreement check involving all parameter blocks.

The amount of reverse work depends on both graph size and primitive costs. Matrix multiplication is not a unit-cost scalar operation. The prototype retains primal arrays throughout a plan execution and does not implement an optimal memory schedule. Its graph-based structure makes checkpointing possible in principle, but no checkpointing performance result is claimed.

The distinction between a cotangent and a parameter update remains explicit. Reverse differentiation computes information about a fixed parameter version. An optimizer subsequently consumes that information to create a new version, closing a separate feedback loop.

The basic consistency relation between the two transformations is

$$\langle \bar{\theta}, v \rangle = \sum_i \langle \bar{\theta}_i, v_i \rangle = DL(\theta)[v].$$

The inner sums use the tensor Frobenius pairing, including vectors and scalars as special cases. The test suite evaluates both sides for the complete recurrent program. Its captured-parameter check is separate: it compares a derivative with respect to a formal parameter against the same derivative while an equal-valued captured parameter is held fixed. This guards a subtle lexical dependency boundary that a primal-value comparison alone would miss.

26. When differentiation commutes with a flush

Pure rebuffering preserves the derivative when its payload map is the identity. More generally, for a smooth conversion $u : A \to B$ between buffers, the tangent route must act by

$$(a, \dot{a}) \longmapsto (u(a), Du(a)[\dot{a}]).$$

Using the same raw tangent tensor after a nonidentity conversion would be incorrect. Reshaping by a fixed coordinate isomorphism and changing units by a scale factor already require their corresponding linear actions on tangents and cotangents.

Let F_p denote execution of a fixed tensor plan p as a function of its numerical inputs. The desired compatibility for forward compilation is

$$\mathsf{run}(p^D, a, v) = \big(F_p(a), DF_p(a)[v]\big)$$

in the ideal real interpretation. The reverse plan has the corresponding Euclidean pairing property. These equations include every route node that affects the payload; a label forgotten by the numerical interpretation does not introduce a derivative of its own.

A higher algebraic extension requires additional structure. If a tangent construction $\mathcal{D}$ is defined on the chosen smooth carriers, assume a compatible transformed program functor T^D and comparison $\sigma_A : \mathcal{D}(TA) \to T^D(\mathcal{D}A)$. A transformed execution algebra α^D would need

$$\mathcal{D}\alpha = \alpha^D \circ \sigma_A.$$

This is a compatibility requirement, with its higher coherences, not a construction automatically available for every higher monad. In particular, a derivative of an arbitrary discrete code representation has not been postulated.

External effects create an additional boundary. Writing a tensor to a file and reading it back can preserve its numerical value while losing the derivative provenance. A system can preserve gradients only by transporting suitable dependency information, a saved derivative program, or a justified recomputation protocol. Neither a filename nor a storage receipt supplies that information alone.

The prototype implements the pure identity case. Rebuffer nodes return their input tensor handles, and generated derivatives pass through them. The outer dispatch can execute an entire differentiated sequence before the host observes its scalar loss. No mandatory host synchronization is inserted at every recurrent route.

This separation explains how buffering can support training without changing the mathematical objective. Physical dispatch boundaries can vary while the logical graph retains the same dependence. Deleting that dependence is a source transformation with mathematical consequences, even when it reduces memory use.

In a pipeline with several pure routes u_i, the derivative program must compose their tangent actions in the same order as their payload actions. Reverse transport uses the adjoints in reverse order. This distinguishes legitimate changes of buffer representation from changes of the differentiated function. A compiler can erase an identity route in the payload graph while retaining its label in a trace, provided the trace interpretation is handled separately.

27. A recurrent neural program across buffered signals

The executable model receives a batch of twelve input signals $X_t \in \mathbb{R}^{48\times 5}$ and an initial hidden buffer $H_0 \in \mathbb{R}^{48\times 24}$. Its shared parameter product is

$$\Theta = \mathbb{R}^{5\times 24} \times \mathbb{R}^{24\times 24} \times \mathbb{R}^{24\times 3} \times \mathbb{R}^{24} \times \mathbb{R}^{3},$$
$$\theta = (W_x, W_h, W_o, b_h, b_o).$$

There are $120 + 576 + 72 + 24 + 3 = 795$ trainable scalar coordinates. The recurrent computation is

$$H_{t+1} = \mathsf{rebuffer}_{t+1}\left(\tanh(X_t W_x + H_t W_h + \mathbf{1} b_h^{\mathsf{T}})\right), \quad 0 \le t < 12.$$

The route preserves the hidden payload and makes it available to the next cycle. The signal sequence is supplied through a product of statically shaped buffers; the runtime does not infer sequence length from untyped data.

The terminal output and scalar objective are

$$Y = H_{12} W_o + \mathbf{1} b_o^{\mathsf{T}}, \qquad L(\theta) = \frac{1}{48 \cdot 3} \|Y - Y^{\star}\|_F^2.$$

The target $Y^{\star}$ is generated by an independently evaluated fixed teacher recurrence of the same dimensions. This is a controlled training workload, not an empirical claim about a real-world dataset. The source program does not call a neural-network layer library.

Each recurrence step is expressed using the higher-order composition combinator from page 22, a source lambda for the affine state-dependent map, and a source lambda for the activation. Products provide the parameter bundle and input sequence; let bindings retain each hidden state. The loss is expressed with multiplication and mean primitives. The source gradient form transforms this complete loss function.

Thus the example exercises several features simultaneously: lexical captures of shared parameters, function-valued source arguments, nonlinear state feedback, repeated buffer reuse, reduction to a scalar objective, and accumulation of gradients into parameters used at every time step. The directional form additionally compiles a sensitivity of the same full objective.

The model is deliberately large enough to make a mistaken buffer cut or shared-weight derivative matter, while small enough to reproduce on a CPU without specialized hardware. The optional CUDA runtime receives the identical serialized tensor plan. Its availability extends the execution target; the evidence in this manuscript remains the measured CPU run.

The parameter sharing is structural. A single product leaf represents W_h and is captured by every affine source lambda; the reverse transform consequently accumulates twelve contributions into that leaf's cotangent. Each hidden-state let binding, in contrast, denotes a different time-indexed value. Sharing weights across time and reusing a state representation are therefore separate mechanisms, even though both use graph references in the implementation.

28. Backpropagation through feedback and the cut problem

Write the recurrent transition as $h_{t+1} = F_\theta(h_t, x_t)$ and the terminal objective as $\ell_\theta(h_T)$. In a coordinate representation, reverse accumulation has the recurrence

$$\bar{h}_T = \partial_h \ell_\theta(h_T), \qquad \bar{h}_t = (\partial_h F_\theta(h_t, x_t))^\mathsf{T} \bar{h}_{t+1}.$$

The parameter cotangent includes a contribution from every cycle:

$$\bar{\theta} = \partial_\theta \ell_\theta(h_T) + \sum_{t=0}^{T-1} (\partial_\theta F_\theta(h_t, x_t))^\mathsf{T} \bar{h}_{t+1}.$$

If an initial state depends on θ, its derivative contributes as well. In the supplied experiment the initial state is held fixed.

A rebuffer route with identity payload and identity derivative preserves these equations. A stop-gradient route has the same primal value but a zero derivative across its boundary. Cutting at time k consequently removes contributions that depend on the earlier history through h_k. The resulting update may be useful as truncated backpropagation, but it is not the full gradient of the original finite-horizon loss.

This distinction is independent of whether the route copies bytes. A CPU copy, GPU view, or immutable reference can all realize the same identity dependence when the differentiation system records it. Conversely, serializing a value and later treating it as a fresh input can sever the dependence even if its bits are unchanged. Provenance and value must therefore be addressed separately in a training runtime.

Full unfolding over T cycles gives a finite differentiable graph. Its memory requirement generally grows with the retained primal states and intermediates needed by the reverse pass. Checkpointing can replace some retention by recomputation when the relevant computations are pure and replayable. It must preserve random choices, parameter versions, and input values to reconstruct the same mathematical trajectory.

Infinite-horizon feedback would require an additional analytic argument. For a stable differentiable fixed point, implicit differentiation can involve an inverse of $I - \partial_h F_\theta$. Invertibility and convergence are hypotheses, not consequences of the recurrence notation. The artifact makes no such infinite-horizon claim.

Its tested derivative is the full derivative of the twelve-cycle unfolding. That finite horizon is an implementation boundary of a nontrivial neural workload, rather than a collection of elementary finite examples offered in place of the requested feedback model.

For the tanh recurrence, the hidden-state derivative has operator norm bounded by $\|W_h\|_2$ under the Euclidean tensor norm, because the activation derivative is bounded by one. A product across $T - t$ cycles is bounded by $\|W_h\|_2^{T-t}$. This explains why retaining a gradient path does not guarantee a well-conditioned gradient. The buffer semantics ensures that the intended dependence is present; stability and conditioning require additional quantitative control of the transition and its trained parameters.

29. Training as an outer buffer/compute/flush loop

The recurrent state loop runs inside a parameter epoch. The training loop operates on a second state containing θ_v, a data policy, and any optimizer state. For simple gradient descent, its update is

$$g_v = \nabla L_v(\theta_v), \qquad \theta_{v+1} = \theta_v - \eta g_v.$$

Buffering fixes the epoch inputs and parameter version, computation produces the objective and cotangent, and flushing publishes the new parameter version for the next epoch. The two loops have different state types even though both follow the same three-phase pattern.

The artifact uses a fixed training batch, a fixed initial hidden state, and a learning rate $\eta = 0.12$ for 180 updates. Every update consumes the derivative of a snapshot of the current parameter arrays. Replacement creates new arrays rather than mutating a parameter while the plan is reading it. Epoch metadata is retained by the pending execution object, although no distributed version protocol is implemented.

For microbatch accumulation, a weighted loss decomposition is necessary. If a full objective is a mean over N examples and microbatch j contains n_j examples, then

$$\nabla L = \sum_j \frac{n_j}{N} \nabla L_j$$

when the objectives separate in the required way. An unweighted sum of microbatch means changes the scale if their sizes differ. Stateful normalization or interactions between examples can invalidate this simple decomposition.

Accumulating several gradients before an optimizer update is also different from updating after every microbatch. Later gradients in the second procedure are evaluated at changed parameters. A buffer/flush optimization that merges these update boundaries therefore needs an algorithmic justification; equal total input data is insufficient.

If an optimizer has momentum or adaptive moments, those tensors belong to the retained state. A replayed update can then alter both parameters and optimizer state twice. A reliable system must distinguish a newly computed gradient from a previously committed update. This issue is not solved by caching the forward loss.

The example uses the simplest outer loop so that its compilation and derivative behavior remain inspectable. It demonstrates optimization steps and observed loss reduction, but does not claim global convergence, generalization, or an optimal schedule. These would require different assumptions and evidence.

A standard conditional descent bound makes the outer update precise. If L has L_s-Lipschitz gradient on the update segment, then

$$L(\theta - \eta g) \le L(\theta) - \eta(1 - \eta L_s/2)\|g\|^2, \qquad g = \nabla L(\theta).$$

The condition $0 < \eta < 2/L_s$ gives a nonincreasing bound there. No suitable global L_s is established for the experiment; its recorded decrease is empirical evidence. The formula identifies the analytic premise that would turn a parameter-update cycle into a descent guarantee.

30. From source terms to executable tensor plans

The compiler first checks source types, including matrix dimensions and the shape of the differentiation argument. It then specializes known source function values in lexical environments. A tensor-valued primitive emits a graph node; a product becomes a tree of graph references; a let binding shares its lowered value. Source function values disappear before the final plan crosses the runtime boundary.

Each graph node contains an operation tag, indices of earlier operands, an output shape, and small attributes such as a route label or constant. Inputs have names determined by their product paths. The ordering invariant is

$$\text{operand index of node } n < n.$$

This makes one-pass execution possible. The graph is a typed single-assignment representation in the compiler; the JSON format records its shapes but is not a complete independently verified type certificate.

When an AD form is encountered, the compiler specializes its function at fresh formal aliases and builds the required tangent or cotangent nodes. The generated reverse graph uses transpose, reduction, and broadcast operations in addition to the original neural vocabulary. The final training plan therefore includes both primal and derivative work. No differentiation is performed by the backend.

The plan executor interprets these tags with a backend interface: array creation, full constants, matrix multiplication through the array operator, hyperbolic tangent, broadcasting, reductions, cloning, and synchronization. The NumPy backend uses CPU arrays. The optional PyTorch backend creates tensors on a selected CUDA device and invokes the corresponding tensor operations. This is a backend-dispatched tensor compiler, not a compiler that emits custom native CUDA kernels.

Serialization enables a plan to be generated once and executed later. The artifact includes a round-trip check that preserves the complete training result. Plans from untrusted sources require additional validation of tags, attributes, shapes, sizes, and resource bounds; the provided reader is intended for trusted local artifacts.

This pipeline realizes the motivating interaction concretely. Code determines the specialized plan, buffered data provide named tensor leaves, computation executes the plan, and routed states feed later nodes. The higher-monadic account supplies the compositional specification above this ordinary graph representation; it is not claimed that JSON serialization carries all higher witnesses.

An ideal plan interpretation assigns each node a value ν_i:

$$\nu_i = \begin{cases} \xi(n_i), & i \text{ is an input named } n_i, \\ [\![\mathsf{op}_i]\!](\nu_{j_1}, \ldots, \nu_{j_r}), & i \text{ has operands } j_1, \ldots, j_r < i. \end{cases}$$

The output tree selects the indicated ν_i and rebuilds the source product structure. This is the ordinary semantic target for specialization. A backend replaces each ideal primitive by a library operation while preserving shapes and dependency order.

The plan need not be minimal. Unused constants or repeated forward subgraphs can remain present without changing the pure output denotation. Removing them is an optimization problem with its own resource and trace considerations. The recorded node count describes the exact transparent plan emitted by this prototype.

31. GPU dispatch, completion, and observation

A CUDA call can return before its device work has completed. Streams order operations within a stream, and dependencies between streams require appropriate coordination [N26]. PyTorch exposes these asynchronous semantics at the tensor level [P26]. A buffer/compute/flush interface must therefore say whether flushing means submitting work, obtaining a result handle, or establishing completion.

The prototype distinguishes these states explicitly:

$$\mathsf{buffered} \longrightarrow \mathsf{enqueued} \longrightarrow \mathsf{completed}, \qquad \text{with a separate } \mathsf{failed} \text{ state.}$$

On the NumPy backend, execution completes synchronously. On the CUDA backend, the outer flush returns device tensor handles after dispatch. The await operation calls device synchronization before asserting completion. A scalar observation transfers the scalar to the host.

Logical rebuffer markers inside the plan do not force a synchronization. They preserve graph dependencies, so later operations can consume the routed tensor in stream order. Inserting a host wait after every hidden state would be a scheduling choice, not a consequence of the source feedback equation. Conversely, using a tensor on a different stream without the required dependency would violate the runtime contract.

The supplied backend uses the current single-stream execution discipline of its process and performs no explicit multi-stream scheduling. Inputs are cloned when a pending computation is created. The single-owner contract excludes concurrent mutations from other streams or host threads. Within that contract, the snapshot separates later parameter replacement from the values used by a queued computation.

Errors may become visible at synchronization rather than at the particular dispatch that caused them. The pending object marks itself failed when an explicit await fails. Its cached handles cannot then be treated as evidence of a successful computation. No rollback of already submitted device work is implied.

Timing also depends on the boundary being measured. Host submission time does not measure completed GPU work. A backend comparison would need warm-up, synchronization or device events, controlled shapes, a declared precision policy, and repeated measurements. None is supplied as a GPU performance result here.

This distinction supports the paper's main argument: feedback is a logical data dependency, whereas dispatch and waiting are physical execution boundaries. A sound implementation can preserve the former while choosing the latter to suit the runtime. Durable storage is a further contract and is not established by GPU completion.

The route receipts in the prototype record the status at the time a marker is encountered. On CUDA they therefore say enqueued, even if a later device-wide await completes the outer pending computation. They are submission records, not independently signed completion receipts. A finer event-based runtime could update a separate completion record for each route. The distinction prevents a log message about reaching a graph node from being interpreted as a durability or device-completion guarantee.

32. Memory, liveness, and checkpointed feedback

A buffer model should account for lifetimes as well as values. A tensor can be released only after its forward consumers and any required reverse computations have finished using it. A topological graph order alone does not identify that last use. A practical memory scheduler requires use counts, alias information, and completion dependencies for asynchronously executing kernels.

The reference executor retains its node values in an array for the duration of one plan run. This makes the implementation transparent and keeps required primal values available to derivative nodes. It is not memory optimal. Snapshots of input arrays introduce additional memory and copying cost, and duplicated forward computations in the training plan can retain more intermediates than necessary.

For a recurrent horizon T, storing hidden states alone has size proportional to Tbh for batch size b and hidden width h. Other intermediates and parameter cotangents contribute further terms. The exact peak depends on schedule and aliasing. A count of logical rebuffer markers does not equal a count of allocated physical buffers: the prototype's markers are identity references.

Checkpointing saves selected states and recomputes intervening portions during the reverse pass. For a pure deterministic recurrence, this can preserve the derivative while trading time for memory. Replay must use the same input sequence, parameter version, initial state, and random choices. If an external read can change between executions, recomputation is not automatically equivalent to using the saved primal.

A route to persistent storage can support larger checkpoints, but its contract must identify what is saved. Values alone may be sufficient when a complete plan and immutable inputs permit justified replay. Otherwise derivative provenance or additional state may be needed. The cost of serialization, transfer, and recovery belongs in a resource analysis; it is absent from a pure denotational identity.

An extended type system could annotate buffers with lifetimes and distinguish borrowed views from owned allocations. A directed refinement could then certify a release schedule while a value interpretation forgets physical addresses. Such a certificate would need to account for asynchronous completion, rather than treating host scope exit as immediate device last use.

The artifact implements none of these advanced memory passes. It exposes a finite dependency graph on which they could operate. Its measured training run establishes that the chosen workload executes in the current CPU environment; it does not establish a general memory bound for arbitrary generated networks.

In a synchronous graph schedule, an allocation candidate has a liveness interval from creation to its final consumer. Shared views extend the underlying allocation's interval to include all aliases. In an asynchronous schedule the final consumer also needs a completion event: dispatching that consumer does not immediately make the input storage reusable.

These constraints suggest a resource interpretation over dependency graphs rather than over source terms alone. A legal allocation assignment maps simultaneously live values to compatible storage without destructive interference. Its value-correctness witness should accompany the mapping, while a separate peak-memory calculation accounts for the chosen schedule and input snapshots.

33. Failures, retries, and distributed routing

Feedback systems often retry an operation after a timeout. A timeout does not reveal whether a remote flush has taken effect. If a storage write or optimizer update was applied before its acknowledgement was lost, repeating it can duplicate the effect. Buffering and monadic sequencing alone do not solve this ambiguity.

A distributed route therefore needs an operation identity, a parameter or state version, and a receiver contract. A receiver can record deduplication state and return an existing receipt for a repeated identity. Atomicity between that record and the intended effect is an additional requirement. A sender-side cache without a corresponding receiver mechanism cannot establish exactly-once application.

The semantics can retain a trace containing submission and completion events:

$$\mathsf{submit}(u, v), \quad \mathsf{accept}(u), \quad \mathsf{commit}(u, v').$$

These events have different evidential force. A valid invariant might state that two successful commits for the same update identity have the same resulting version. Whether that invariant holds depends on the actual protocol and storage transaction semantics.

For neural training, a stale gradient can be mathematically well defined at its old parameters while being inappropriate for an update that expects current parameters. An asynchronous optimization algorithm may intentionally accept it, but then the algorithm must specify that policy. A version check that rejects stale work and an algorithm that tolerates stale work are different feedback systems.

The prototype is single-process and single-owner. One pending object performs at most one successful dispatch through its own interface. Repeated flush calls return cached handles. If dispatch raises an exception, the object is poisoned and refuses an implicit retry. This local rule prevents accidental resubmission through that object; it is not a distributed deduplication protocol or crash-recovery mechanism.

A failed dispatch can have produced partial work before reporting an error. The implementation neither claims rollback nor attempts to reuse a partially completed graph. An operator who wants to retry must construct a new computation with a suitable application-level policy. The general semantic model leaves room for a stronger protocol, but does not attribute that protocol to the current artifact.

These distinctions give flush a useful precise meaning: it is an effectful routing request with a declared outcome interface. Its reliability follows from the chosen implementation and environment, rather than from a formal identification with program substitution.

A cached result also relies on ownership. The pending object returns the same handles after successful dispatch; a client that mutates those tensors changes what later clients observe. The intended contract therefore treats returned payloads as immutable while they are shared. This is a convention enforced by the application's discipline, not by a hardware write-protection mechanism in the prototype.

For a distributed implementation, ownership and version information would have to cross the route interface as explicit protocol data. A pointer identity or a process-local cache key cannot serve as a globally meaningful update identity after a restart.

34. Numerical semantics, error, and cost

The equations for the neural recurrence are smooth real equations. An executable backend uses floating-point arrays and library kernels. Reassociation, fused operations, different reduction orders, and precision changes can alter numerical outputs. A compiler should therefore distinguish exact denotational preservation from preservation within a quantified error contract.

For a recurrence with per-step error δ_t and a local Lipschitz bound L_t in the retained state, a schematic propagation bound is

$$e_{t+1} \leq L_t e_t + \delta_t.$$

Unfolding gives

$$e_T \leq \left(\prod_{j=0}^{T-1} L_j \right) e_0 + \sum_{k=0}^{T-1} \delta_k \prod_{j=k+1}^{T-1} L_j.$$

The bounds require compatible norms and valid hypotheses on the evaluated region. They show why feedback can amplify small local differences; they do not supply numerical constants for an arbitrary trained network.

Derivative errors need their own analysis. Agreement between a forward directional derivative and a reverse gradient contraction checks two implementations of the same local derivative, but both could share a systematic mistake. A central-difference comparison provides additional independent evidence, subject to truncation and rounding error. One tested direction does not prove every component correct for every possible input.

The reference experiment uses float64 on NumPy. The optional Torch backend supports float64 and float32. Its GPU path has not been executed in this environment. No conclusion about mixed precision, Tensor Cores, accelerator throughput, or cross-device bitwise reproducibility follows from the CPU results.

Cost also has several meanings. A symbolic node count measures the representation, not elapsed time. A matrix multiplication has a cost that depends on its dimensions and backend. A logical flush can be an identity reference, a device transfer, or a storage transaction, with very different physical costs. A useful cost semantics must attach the appropriate cost interpretation to each route.

The evidence file records one local training time and compilation time for reproducibility. These are observations of one process, including the stated snapshot and synchronization behavior. They are not a benchmark or evidence that the buffer/compute/flush organization is faster than established frameworks. The demonstrated result is successful compilation and training with explicit route semantics.

The same error discipline applies to optimization certificates. If pass one changes the denotation by at most ε_1 and pass two amplifies its input error by at most L while adding ε_2, the composed bound is $L\varepsilon_1 + \varepsilon_2$. Certificates at different observation norms require a conversion bound. This is a quantitative analogue of carrying the observation interface through higher coherence: an estimate is useful only together with the space and norm in which it is stated.

35. The executable artifact and its trust boundary

The publication package contains the source manuscript, a small Python compiler, two tensor backends, the recurrent program generator, tests, and recorded evidence. Its central files have distinct responsibilities:

Path	Role
`bcf/core.py`	Types, source AST, specialization, explicit AD, plans, dispatch state.
`bcf/train.py`	Recurrent lambda term, controlled data, epoch updates, CLI.
`tests/` `test_compiler.py`	Neural integration, derivative, snapshot, failure, and serialization checks.
`evidence/`	CPU training record, test output, and generated tensor plan.
`paper/`	Editable LaTeX source and final manuscript.

NumPy is the required numerical dependency. PyTorch is optional and is loaded only when its backend is requested. CUDA execution requires an appropriate PyTorch installation and a working CUDA-capable device. The backend fails explicitly if CUDA is requested but unavailable. It does not silently relabel CPU work as GPU execution.

The trust boundary includes the Python interpreter, NumPy or PyTorch, their native libraries, the device runtime when used, and the host environment. The compiler does not execute source strings with `eval`. It accepts a constructed source AST, checks its admitted types, and emits a trusted local plan. A secure network service accepting arbitrary plan JSON would require additional validation and resource isolation.

The mathematical source language is pure at the tensor level. Its route markers describe logical state reuse. Real storage and network effects are discussed by the general semantics but are not implemented as hidden behavior in the training program. This makes it possible to inspect how a source hidden state becomes a graph reference and how its derivative is propagated.

The artifact is a prototype compiler and runtime, not a formally verified Lean-to-Rust toolchain. The requested executable capability is supplied through another runtime with a CUDA backend. No Rust binary is included or claimed. A Rust implementation could consume the same admitted operation vocabulary after defining a compatible backend and independently validating its numerical and ownership semantics.

Reproducibility is supported by fixed dimensions, a fixed random seed, explicit hyperparameters, readable code, and saved results. It is not guaranteed by the package name alone. Dependencies and hardware remain part of any reproduced experiment's report.

The generated graph is intentionally portable at the operation level. A new backend must implement the same scalar constants, transpose, broadcast, and reduction conventions as well as matrix arithmetic. Matching only the forward matmul and tanh operations is insufficient for training, because the compiled reverse plan uses the additional operations to form weight and bias cotangents.

36. Reproducing the compiled neural training program

The package runs from its root with Python 3.10 or later; the recorded environment used Python 3.12.14. Create an isolated environment and install the local package with its NumPy dependency. The following commands run the full recurrent workload and its integration checks, not a substitute arithmetic demonstration:

```
python3 -m venv .venv
. .venv/bin/activate
python -m pip install -e .
python -m unittest discover -s tests -v
python -m bcf.train --backend numpy \
  --output run-numpy.json --plan recurrent-plan.json
```

The source program is assembled and typed before execution. The `-plan` option exports its training plan, including the generated reverse operations. The `-output` option records dimensions, seed, hyperparameters, loss history, graph statistics, runtime identity, and whether CUDA was actually used. Output paths are ordinary files and are not claimed to have a durable transactional commit contract.

For GPU execution, install a CUDA-capable PyTorch distribution suitable for the local platform using the official installation selector. Then run the same workload through the optional tensor backend:

```
python -m bcf.train --backend cuda --dtype float64 \
  --output run-cuda.json --plan recurrent-plan.json
```

The backend checks CUDA availability and reports an error when the requested execution target is absent. Float32 is an explicit alternate option. A `torch-cpu` mode is available for checking the optional backend independently of a GPU.

The optional integration check compares the compiled CUDA training outputs with NumPy when PyTorch and a CUDA device are available. In the present environment it was skipped because PyTorch was absent. Consequently, the GPU command is a supported execution path in the source package, not a command reported as successfully executed here.

Reproduction should retain the entire signal sequence and all five parameter blocks. Changing the horizon, batch size, seed, precision, or loss normalization changes the experiment. The default generator constructs a teacher target and a separately initialized student, then optimizes the student's terminal loss.

The test suite and evidence files should travel with the manuscript. They allow a reader to connect a mathematical claim about gradient transport to the exact program and environment used for the reported run. They do not replace the general semantic proof obligations stated earlier.

For independent review, retain both the generated plan and the source that produced it. A plan alone can reproduce numerical execution without demonstrating that source specialization was correct. The source alone can demonstrate the intended term without recording the actual emitted operations. Their agreement, together with the test output and backend identity, forms the reproducibility record.

37. Recorded evidence and its interpretation

The local reference run used the NumPy CPU backend in float64, Python 3.12.14, and NumPy 2.3.5. The seed was 314159; the recurrent horizon was twelve; the batch size was 48; the hidden width was 24; and the optimizer used 180 gradient-descent updates with learning rate 0.12. The resulting loss history was:

Epoch	Mean squared loss	Fraction of initial loss
0	0.05656894568	1.00000
30	0.00689270725	0.12185
60	0.00417760715	0.07385
90	0.00342413699	0.06053
120	0.00301851476	0.05336
150	0.00273023763	0.04826
180	0.00250650652	0.04431

The final loss is approximately 4.431% of the initial loss, a reduction of approximately 95.569%. This is training loss on the controlled fixed batch. No held-out accuracy or generalization measurement is reported.

The training plan contains 463 nodes, including 106 matrix multiplications, 24 hyperbolic-tangent nodes, and 24 rebuffer markers. There are twelve logical state routes in one forward recurrence. The plan evaluates the forward objective separately from the forward graph used to generate its gradient, accounting for the two sets of twelve markers. Node counts are representation statistics, not hardware kernel counts or speed measurements.

Nine integration checks passed. They cover the higher-order source recurrence against an independent eager evaluation; agreement among compiled JVP, reverse-gradient contraction, and a central difference; constant treatment of a captured parameter during partial differentiation; recurrent gradient dependence; plan serialization; input snapshot isolation and cached flushing; refusal of an implicit retry after failure; source shape rejection; and loss reduction. One additional CUDA parity check was skipped because PyTorch was not installed.

The derivative comparison uses a direction spanning all five parameter blocks of the 795-dimensional parameter space. The central-difference step is 10^{-5}. The reverse contraction and compiled JVP are compared at a tighter tolerance than the finite-difference comparison. These choices are recorded in the executable tests rather than left as unspecified numerical assertions.

The evidence supports successful execution of this nontrivial fragment. It does not establish the higher-monad construction for all source languages, prove a compiler correct for every input, or validate the unexecuted GPU backend. Those boundaries are part of the result.

The raw derivative record gives a compiled directional derivative of approximately 0.00768303920. Its absolute difference from the reverse pairing is 1.73×10^{-18}, and the reverse pairing differs from the central difference by 1.78×10^{-13}. The full precision values are saved separately. The record should be read with the seed and direction generation rule. It is a local sensitivity of the initial student parameters, not the gradient norm or a sensitivity measured after the final training update.

38. Open obligations, conclusions, and authorship

The framework identifies several substantive next steps. A formal development should construct the chosen binding-aware higher program monad, its phase or version indices, and its execution interpretation. It should then connect the executable source syntax to that construction. Giving a name to a monad is not a substitute for constructing its operations and coherent laws in a selected model.

A verified compiler extension should formalize source typing, specialization, derivative rules, and route preservation. The mathematical difficulty is especially clear at the boundary between higher-order source functions and first-order tensor graphs. The present artifact is inspectable and tested, but it supplies no Lean proof, no alternate-kernel verification, and no certified native code generator.

The runtime extension should add liveness-based memory reuse, checkpointing with explicit replay conditions, and a validated plan interchange format. GPU work should first exercise the existing backend and parity check, then measure completed work under a controlled protocol. A Rust backend is a reasonable engineering target, but it must preserve shape, ownership, and numerical contracts rather than inheriting correctness from the Python design.

For infinite feedback, a selected guarded or analytic model is needed. For distributed flushing, explicit receipt and recovery protocols are needed. For a higher-topos formulation of local execution, descent and base-change hypotheses must be supplied. These are distinct obligations; solving one does not solve the others.

Conclusion. Buffer/compute/flush is a compositional feedback organization: signals become available in buffers, code acts on them with retained state, and outputs are routed to storage or further computation. Evaluation explains the general code–data interaction, while literal convolution appears under explicit kernel assumptions. Higher monads organize substitution of program structure; execution algebras and effectful interpretations realize it; guarded feedback controls repetition. Differentiable routing preserves the chain rule when it preserves the required dependence.

The recurrent compiler demonstrates that these distinctions can inform a working training system. Its routes are visible in the generated graph, its derivatives are explicit programs, and its epoch updates close an outer feedback loop. The measured CPU experiment establishes compilation and training for that fragment, with an optional unverified CUDA execution path supplied for further evaluation.

Authorship and automation. The anonymous byline is intended for review and asserts no institutional affiliation. This draft and its implementation were prepared with AI assistance at the commissioning researcher's direction. Human review and responsibility for any submitted version remain necessary. No independent referee review, journal acceptance, or public submission is represented as having occurred.

A practical next milestone is a preservation development for precisely the admitted source fragment and its tensor plan, with shape-indexed primitives and explicit derivative boundaries. This would give a concrete bridge between the abstract semantic contract and the existing implementation. A second milestone is a GPU reproduction of the same graph with reported parity, precision, and completed execution timing. Keeping these milestones distinct makes it possible to assess progress without treating a successful numerical run as a substitute for a general semantic proof.

39. References: computation, higher algebra, and feedback

References

[M91] Eugenio Moggi. *Notions of computation and monads*. Information and Computation 93(1), 55–92, 1991. Original article (PDF).

The foundational distinction between values and effectful computations motivates the separate runtime interpretation used here. The present manuscript applies that distinction to repeated buffering and routing; it does not identify every flush with the multiplication of a monad.

[L17] Jacob Lurie. *Higher Algebra*. September 18, 2017, especially Section 4.7. https://www.math.ias.edu/~lurie/papers/HA.pdf.

This is the reference for monads as associative algebra objects in endofunctors and for the accompanying higher coherence. Our program signature, execution choices, and implementation contract are additional proposed data, rather than constructions automatically provided by that general definition.

[L09] Jacob Lurie. *Higher Topos Theory*. Annals of Mathematics Studies 170, Princeton University Press, 2009. https://www.math.ias.edu/~lurie/papers/HTT.pdf.

Higher sheaves and higher-topos structure provide the setting for the conditional contextual interpretation on pages 18–19. No claim of automatic descent for executable code or automatic smooth differentiation is attributed to the ambient topos axioms.

[PP03] Gordon Plotkin and John Power. *Algebraic operations and generic effects*. Applied Categorical Structures 11, 69–94, 2003. Author manuscript (PDF).

This supports the distinction between operation signatures and interpretations of effects. The interaction between buffering, state, failure, and external storage still requires an explicit effect theory or a compatible interpretation in the present model.

[JSV96] André Joyal, Ross Street, and Dominic Verity. *Traced monoidal categories*. Mathematical Proceedings of the Cambridge Philosophical Society 119(3), 447–468, 1996. doi:10.1017/S0305004100074338.

Trace provides an abstract language of feedback under its axioms. The causal recurrence in this manuscript does not assume that every smooth instantaneous feedback equation has a total trace.

[B12] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. *First steps in synthetic guarded domain theory: step-indexing in the topos of trees*. Logical Methods in Computer Science 8(4:1), 2012. https://arxiv.org/abs/1208.3596.

The guarded account motivates delayed state feedback. Its fixed-point structure is treated as additional model structure, not as a property of every higher topos.

40. References: differentiation and accelerator execution

References

[ER03] Thomas Ehrhard and Laurent Regnier. *The differential lambda-calculus*. Theoretical Computer Science 309, 1–41, 2003. doi:10.1016/S0304-3975(03)00392-X.

This is the classical differential lambda-calculus reference. Our executable fragment is a restricted smooth typed language with directional and scalar-gradient forms, rather than an implementation of the whole algebraic calculus.

[HSV20] Mathieu Huot, Sam Staton, and Matthijs Vákár. *Correctness of automatic differentiation via diffeologies and categorical gluing*. FoSSaCS, 2020. https://arxiv.org/abs/2001.02209.

This supplies a semantic route to higher-order differentiable programming. The new compiler's specific preservation obligations are not claimed to follow without connecting its syntax and implementation to that framework.

[V21] Matthijs Vákár. *Reverse AD at higher types: pure, principled and denotationally correct*. ESOP, 2021. https://arxiv.org/abs/2007.05283.

Higher-type reverse differentiation needs more structure than the first-order tensor reverse pass implemented here. The prototype specializes higher-order functions before deriving the tensor gradient.

[C20] Robin Cockett, Geoffrey Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon Plotkin, and Dorette Pronk. *Reverse derivative categories*. CSL, LIPIcs 152, article 18, 2020. https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2020.18.

The reverse operation is given an abstract categorical axiomatization in this work. Our concrete cotangent rules use finite-dimensional Euclidean tensor representations.

[N26] NVIDIA. *CUDA Programming Guide: Asynchronous Execution*, Section 2.5. Documentation consulted November 3, 2026. Official asynchronous execution documentation.

This supports the distinction between dispatch, stream ordering, and synchronization. The manuscript reports no GPU performance measurement and no proof of a custom CUDA scheduler.

[P26] PyTorch. *CUDA semantics*. Official documentation, consulted November 3, 2026. https://docs.pytorch.org/docs/main/notes/cuda.html.

The optional backend relies on PyTorch tensor dispatch and its device synchronization interface. Its source is included; GPU execution was unavailable in the reported environment. Installations should use the official platform-specific PyTorch installation instructions.