MightBe: An Audited Neural Runtime with Hot Structural Mutation and Traceable Multi-Hop Reasoning

yxpil

October 25, 2026

Abstract

MightBe is a single-process system that couples a from-scratch neural-network runtime to a relational store through a learned, auditable index. We make three contributions. First, we specify a hot-mutation protocol for a live network: an ALTER operation is merged into a candidate specification, validated, rebuilt, and re-hydrated with weights migrated by (layer, parameter, shape); the candidate must then pass a smoke forward pass on a zero tensor before it is published, and publication is a lock-free atomic swap of an `ArcSwap`-held snapshot, so readers in flight continue to observe the previous version and any failure leaves the previous version serving unchanged. Second, we give an auditable inference graph in which every answer is a scored path together with a bounded evidence set, and we specify the confidence algebra exactly: a path score is a product of per-step contributions, several paths reaching the same node are combined by a noisy-OR, and a verifier reranks the result by geometric interpolation with a semantic similarity term. Third, we couple the store to the learned index with an idempotent insert trigger that routes text rows through a normalization, tokenization, TF-IDF, TextRank and pointwise-mutual-information pipeline before the association graph is built. We report the complete result of the artifact's own test suite — 32 test targets, 251 tests, all passing, one doc-test ignored — and we state plainly that the repository contains no benchmarks and no accuracy or latency measurements, so we give a measurement protocol rather than results. We also disclose that the test suite itself was generated with the assistance of a large language model, and that six modules named in the project's own documentation have no implementation.

1 Introduction

A neural-network runtime is usually treated as a fixed artifact: it is trained, exported, deployed, and thereafter treated as a black box whose structure can change only by restarting the process. That assumption is convenient, and it is wrong more often than one would like. A model is routinely promoted between environments, a feature is added to a production network while it is serving traffic, a hyper-parameter is corrected after a bad training run, and a data-derived layer must be grafted onto a network that is already handling requests. Each of these is an *update*, and each of them is usually executed by taking the service down.

Alongside the runtime sits a second system: a store that holds the data the network was trained on, and a reasoning layer that answers questions about that data with an explanation attached. The hard part of such a system is not any individual component — automatic differentiation, a page store, a keyword index are all well understood — but the seams between them. A question that is answered by a learned index must be able to say where its answer came from, and a network that is mutated must be able to say which version answered a request.

MightBe is a single-process Rust workspace of nine crates that attempts to make those seams explicit. It is not a production database and does not present itself as one: it has no join optimiser, it isolates at read uncommitted, it keeps its undo log in memory, and it exposes its service over a line-oriented TCP protocol rather than a wire-compatible MySQL protocol. Its value, if it has one, is that it makes three specific mechanisms auditable end to end, and that it is small enough to read.

This paper makes three contributions.

1. **An audited hot-mutation protocol for a live network** (Section 3.1). We give a six-stage protocol whose every stage is checkable: candidate merge, specification validation, rebuild, weight migration keyed by (layer, parameter, shape), a smoke forward pass on a zero tensor, and a lock-free snapshot swap. We show that the protocol is failure-atomic in a precise sense: if any stage fails, the previously serving layer set, the version counter, and the optimiser state are all left exactly as they were, and the caller receives an error.

2. **An auditable inference graph with an exactly specified confidence algebra** (Section 3.2). Every answer is a scored path through a typed graph plus a bounded evidence set. We give the algebra in closed form — per-step contribution, path score, noisy-OR over paths, verifier rerank, temperature calibration — and we note that one specific path score in the artifact is pinned by an integration test to a tolerance of 10^{-5}, so the specification is checkable rather than merely descriptive.

3. **A relational store coupled to a learned index by an insert trigger** (Section 3.3). A row landing in a table designated by a network's header is routed by a trigger into a text pipeline and thence into an association graph. The

trigger is idempotent per (table, row id), so the learned index cannot drift out of step with the table by double counting.

We report exactly one measured quantity: the outcome of the full workspace test suite. We report it in full, including the target-by-target breakdown, and we report the wall-clock cost of the slowest target. We do not report accuracy, latency, or throughput, because the repository contains no such measurements; Section 5.2 instead specifies the protocol that would produce them, with placeholders marked, and Section 8 lists what is missing from the artifact altogether. Section 7 positions the work against automatic differentiation engines, probabilistic ranking, and explainable inference.

2 Background

2.1 Autodiff engines and the hot-update problem

Reverse-mode automatic differentiation computes a gradient by propagating partial derivatives backwards from an output to the leaves of a computation graph Rumelhart et al. (1986). In practice an engine represents that graph explicitly. Each node stores an operation, its inputs, the value it produced, and (after the backward pass) the gradient that arrived at it. Different engines differ in how much structure they materialise and how they get from a forward graph to a backward sweep.

MightBe's core uses a deliberately minimal representation, chosen so that the hot-mutation problem has a tractable answer. A tensor is a flat row-major vector of `f32` together with a shape and a stride vector. The graph is an append-only arena: nodes are pushed onto a vector and referred to by a handle, `Var(usize)`. A node's inputs are always handles of strictly smaller index, because a node can only be created after the nodes it consumes. The backward sweep therefore does not need to topologically sort the graph; it walks the arena in reverse index order, and that order is a topological order by construction. The tape has 31 operation variants, covering the elementwise and matrix primitives, the activation functions, the shape operations, one-dimensional convolution and pooling, an embedding lookup, the two loss functions, and a single `Plugin` variant that carries an arbitrary user-supplied backward rule.

The optimisers are equally conventional: stochastic gradient descent, the same with momentum Rumelhart et al. (1986), and Adam Kingma and Ba (2014) with a learning rate of 0.002, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ by default. Gradient clipping is available in the style of Pascanu et al. Pascanu et al. (2013). One detail turns out to matter a great deal for the rest of this paper: the optimiser state is keyed by the *full parameter name*, not by a pointer or an index into a parameter vector. That choice is what makes it possible to mutate a network's structure while it is running without silently discarding the first and second moment estimates of every surviving parameter.

That is the whole of the hot-update problem in this system. If the runtime were immutable, mutation would be a non-issue. Because it is not, we need a protocol that answers four questions:

1. What exactly changes? If we mutate the *structure* of a network — adding a layer, dropping a layer — the surviving parameters may change shape, and the surviving optimiser state may no longer correspond to anything.

2. When does the change become visible? If visibility is not atomic, a concurrent reader can observe a half-built network.

3. How do we know the new network works? A network that compiles and has consistent shapes may still fail at run time, for instance because a recurrent layer was inserted where a tensor rank does not permit it.

4. What happens to the old version? If the change fails, the old version must keep serving; if it succeeds, an answer that was computed under the old version must remain identifiable as such.

Section 3.1 answers these with a protocol that has an explicit failure path, and Section 5.1 reports the integration tests that pin its observable consequences.

2.2 Explainable inference, evidence and calibration

The second subsystem answers questions about stored data by walking a graph of typed nodes. Nodes are words, documents, labels, fields, values and tables; edges are typed as well, and carry a score in $[0, 1]$. The interesting design question is not how to find a path but what to promise about the answer.

Three promises are common in the explainable-AI literature, and they are worth separating because a system can keep one and break another.

Traceability: the answer must be attributable to specific supporting data. In our setting this means that the answer carries a bounded set of document identifiers, and that the set is derived from the path rather than retrieved afterwards by a similarity search.

Confidence: the answer must carry a number that means something, and that number must be computed by a rule the reader can check. It is common for a confidence to be an uninterpreted output of a second model; we instead define it compositionally from the scores along the path.

Calibration: the confidence should agree with the observed frequency of correct answers at that confidence level. The standard tool is the expected calibration error, computed over binned confidence levels Guo et al. (2017); Naeini et al. (2015). Calibration is a property of a deployed system, not of a formula, and it can only be measured against labels; our system therefore implements the calibration machinery but cannot, without labelled data, report a calibrated score — see Section 8.

A fourth property is worth naming because the artifact refuses it: *generation*. The reasoning layer never produces free

Algorithm 1 Hot structural mutation of a live network.

Require: A network N and an alteration *op*

$L \leftarrow \text{changeLock}(N)$ ▷ per-network mutex
$S \leftarrow N.\text{spec}()$ ▷ copy of the live spec
$S \leftarrow \text{Merge}(S, op)$ ▷ add/drop layer, set params, reload
if *op* adds or drops a layer **then**
    $S.\text{version} \leftarrow N.\text{version} + 1$
end if
$\text{Validate}(S)$ ▷ spec-level consistency
$B \leftarrow \text{Build}(S)$ ▷ shape inference, head layer
$(W, \Omega) \leftarrow \text{Migrate}(B, N.\text{dumped}())$
$x \leftarrow \text{Zeros}(S.\text{inputShape}())$
$y \leftarrow \text{Forward}(B, x, \text{training} = \text{false})$ ▷ smoke gate
if y failed **then**
    Err("smoke failed, old version retained")
    return ▷ N is untouched
end if
$N.\text{layers} \leftarrow B$ ▷ atomic replacement of the layer set
$N.\text{history} \leftarrow \text{Push}(N.\text{version}, S_{\text{old}}, W_{\text{old}})$
$N.\text{spec} \leftarrow S$ ▷ and $N.\text{topology} \leftarrow S.\text{topologyJson}()$
$N.\text{version} \leftarrow S.\text{version}$ ▷ release store
$\text{RefreshSnapshot}(N)$ ▷ atomic swap of the snapshot
return $N.\text{version}$

text. Its output is a node, a confidence, a set of paths, and a set of evidence documents. This is a deliberate restriction, stated in the project's own non-goals, and it is the reason the reasoning layer can be audited at all: every claim is a pointer into a graph rather than a string that must be checked against the source.

3 Design

3.1 Audited structural mutation of a live network

A network in MightBe is owned by a registry entry. The entry holds the network's specification (a TOML document describing layers, the head, the training configuration, and the header that binds the network to a source table and a text column), the live layer objects, the optimiser state, a monotonic version counter, a training-step counter, a bounded history of previous versions, and a snapshot pointer. A per-network mutex serialises mutations; a separate lock guards the layer vector itself, which is the only shared mutable state that inference touches.

The protocol is given in Algorithm 1. Its shape is the shape of the argument in the previous subsection: merge, validate, rebuild, migrate, smoke, swap.

Three properties of Algorithm 1 deserve to be stated precisely, because they are what the integration tests check.

Failure atomicity. Every stage before the swap either mutates only local state (the candidate specification, the freshly built layer set, the warning list) or mutates state that is not yet visible. The first mutation of visible state is the assignment of the new layer vector, which happens after the smoke gate has succeeded. If validation, building, migration or the smoke pass fails, the function returns an error and the live network is bit-for-bit unchanged: the same layer objects, the same version number, the same optimiser state, the same snapshot. The integration test `alter_smoke_failure_rolls_back_completely` constructs exactly this situation — it inserts a recurrent layer whose shape requirements are incompatible with the running network, observes the error, and asserts that the version counter did not move and that the old version still answers.

Shape-aware weight migration. Migration is keyed by the full parameter name, for instance `dense1.weight`, and is conditional on the shape. If a parameter survives with the same shape, its old value is copied into the new layer object. If it survives with a different shape, the new initialisation is kept and a warning is emitted, because silently reshaping a learned matrix would be worse than losing it. If it is new, the layer's own initialisation strategy is used, again with a warning. The warnings are returned to the caller rather than logged, so a mutation is never silent.

Atomic publication. A reader of the network's metadata — the version, the topology, the training-step count, the state, the last loss — reads a snapshot object held behind an `ArcSwap`. Publication constructs a new snapshot and stores it; the store is atomic and wait-free for readers, who hold an `Arc` to the snapshot they read. A reader that arrived before the swap keeps observing the old version, and the integration test `snapshot_is_stable_while_altering` asserts precisely that: the old snapshot still reports the old version and the old topology while an alter is in flight.

The protocol also supports undo. Every successful mutation pushes the replaced version — its specification and its dumped weights — onto a bounded in-memory history; consecutive alterations of the same version replace rather than duplicate the history entry, so that `restore version` always finds the most recent state of that version. `Restore` rebuilds from the historical specification and rebinds the historical weights, then bumps the version back.

Figure 1 shows the pipeline and, in dashed red, the failure path that keeps the previous version serving.

3.2 An auditable inference graph

The reasoning layer holds a `ReasonGraph`: typed nodes, typed edges with scores, and an evidence set attached to some nodes. A question is answered by a beam search over the graph, starting from a node that matches the query and returning *conclusions*: nodes that are not merely intermediate words but things the system is willing to answer with. Nodes are grouped into three scopes — word scope covers words, documents and labels; field scope covers fields and values; table scope covers tables — and a configurable scope filter decides which may be traversed. By default only word scope is

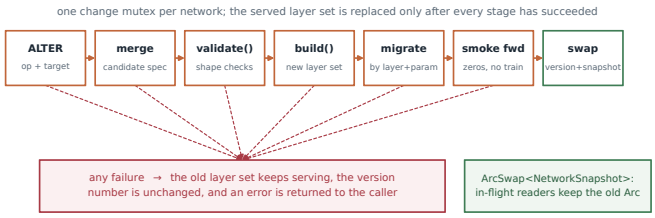


Figure 1: The hot-mutation pipeline. A candidate is merged into a specification, validated, rebuilt, re-hydrated with migrated weights, and put through a smoke forward pass on a zero tensor; only then is the served layer set replaced and the snapshot published. The dashed path is taken if any stage fails.

traversable, and within it a word node is an intermediate step rather than an answer, so the conclusions a default configuration can return are documents and labels. Widening the scope filter admits the structural nodes, at the cost of the cross-scope penalty described below.

Algorithm 2 gives the search and the confidence accumulation.

The algebra that Algorithm 2 implements is worth writing out, because a confidence that cannot be recomputed by hand is not auditable.

Per-step contribution. For an edge e traversed at hop t, the contribution is

$$c_t \;=\; s(e) \cdot \gamma \cdot \mathbb{1}[\mathrm{scope}(v_t) \neq \mathrm{scope}(v_{t-1})] \cdot g, \qquad (1)$$

where $s(e)$ is the edge score, γ is the depth decay (default 0.8), g is the cross-scope gate (default 0.6), applied once per change of scope between the word, field and table scopes. The cross-scope factor is what stops a path from accumulating a large score by oscillating between structural levels of the graph.

Path score. A path of h edges has score

$$p \;=\; \min\left(1,\; \gamma^{h} \cdot \prod_{t=1}^{h} s(e_t)\right), \qquad (2)$$

which is the product of the contributions in Equation 1 with the gate factorised out; when a path crosses scopes, the gate multiplies in once per crossing. A rule prior attached to the graph multiplies the path score by $(1 + \min(c_{\mathrm{rule}}, 1))$, so a rule can strengthen a path but never more than double it, and the result is capped at 1.

Multi-path aggregation. Several paths may reach the same conclusion. They are combined by a noisy-OR,

$$P \;=\; 1 - \prod_{j} (1 - p_j)\,, \qquad (3)$$

which is the standard independence-free combination for noisy evidence: it is monotone in every p_j, it is bounded by 1, and it collapses to p_1 when there is a single path. The independence assumption is deliberately not made — Equation 3 is an aggregation convention, not a probabilistic claim, and the paper says so.

Algorithm 2 Beam search with an explicit confidence algebra.

Require: A graph G, a start node s, options ω

$\mathcal{B} \leftarrow \{\mathrm{PartialPath}(s, 1.0, \emptyset)\}$
$\mathcal{R} \leftarrow \emptyset$ ▷ node $\mapsto$ list of paths
for $h = 1 \ldots \omega.\mathrm{maxHops}$ **do**
  $\mathcal{B}' \leftarrow \emptyset$
  for partial path $\pi \in \mathcal{B}$ **do**
    for neighbour v of $\pi.\mathrm{last}$, via edge e **do**
      if v is already on π **then continue** ▷ cycle pruning
      if $\mathrm{scope}(v) \notin \omega.\mathrm{scopes}$ **then continue**
      if $e.\mathrm{score} < \omega.\mathrm{minEdge}$ **then continue**
      $c \leftarrow e.\mathrm{score} \cdot \omega.\mathrm{depthDecay}$
      if $\mathrm{scope}(v) \neq \mathrm{scope}(\pi.\mathrm{last})$ **then**
        $c \leftarrow c \cdot \omega.\mathrm{crossScopeGate}$
      end if
      push $\mathrm{PartialPath}(\pi, v, \pi.\mathrm{score} \cdot c)$ to $\mathcal{B}'$
    end for
  end for
  sort $\mathcal{B}'$ by score, truncate to $\omega.\mathrm{beamWidth}$
  for each path $\pi \in \mathcal{B}'$ **do**
    $\mathcal{R}[\pi.\mathrm{last}] \leftarrow \mathcal{R}[\pi.\mathrm{last}] \cup \{\pi\}$
  end for
  $\mathcal{B} \leftarrow \mathcal{B}'$
end for
for each node n in $\mathcal{R}$ with $\mathrm{isConclusion}(n)$ **do**
  $P \leftarrow 1 - \prod_j (1 - p_j)$ ▷ noisy-OR over paths
  $E \leftarrow \mathrm{GatherEvidence}(n, \mathcal{R}[n], \omega)$
  emit conclusion $(n, P, \mathcal{R}[n], E)$
end for

Verifier rerank. An optional verifier reranks a conclusion by interpolating, in log space, between the path confidence and a semantic agreement term:

$$c' \;=\; c^{\lambda} \cdot v^{\,1-\lambda}, \qquad (4)$$

with $\lambda = 0.7$ by default and $v = \mathrm{clamp}(\cos(e(q), e(d)), 0, 1)$ the cosine similarity between the encoder of the query and the encoder of the candidate's description, clamped to $[0, 1]$. Because the interpolation is geometric, $c' \leq c$ whenever $v \leq 1$; the verifier can lower a confidence but the term $v^{1-\lambda}$ can also raise it above c when $v = 1$, because $c^{\lambda} > c$ for $c \in (0, 1)$ and $\lambda < 1$. The net effect is a shrinkage towards the middle, which is the intended behaviour when the verifier is a better-calibrated model than the path score and a harmful one when it is not; this is one of the reasons the verifier is switchable.

Calibration. An optional temperature calibration maps a probability p through the logit, divides by a fitted temperature

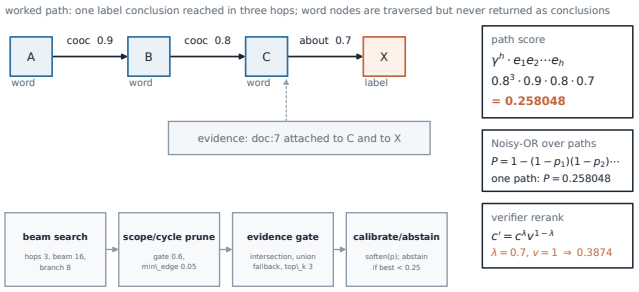


Figure 2: The reasoning graph and the confidence algebra. A path of three edges is scored by Equation 2, several paths reaching one node are combined by Equation 3, and a verifier reranks by Equation 4. Word nodes are traversed but never returned as conclusions.

T, and applies the logistic function again:

$$\text{soften}(p) \;=\; \sigma\!\left(\frac{\text{logit}(p)}{T}\right). \tag{5}$$

The temperature is fitted by minimising the mean negative log-likelihood of a set of labelled observations over a log-spaced grid of T in $[0.05, 20]$ with 200 steps; a fit with fewer than four observations is refused with a dedicated error code. The expected calibration error is computed over ten equal-width bins.

Evidence and abstention. The evidence attached to a conclusion is the node's own evidence when the conclusion is a label or a document; otherwise it is the intersection of the document sets of the individual paths that reached it, falling back to the union when the intersection is empty, truncated to at most `evidence_top_k` $= 3$ documents. If evidence is required and none can be produced, the answer is refused with a distinct error code rather than returned without support. Independently, the system abstains when the best remaining confidence falls below a threshold (default 0.25), again with a distinct error code. Abstention and refusal are different: refusal means the answer could not be supported, abstention means the answer could not be supported *confidently enough*.

A checked instance. Consider the graph used by the artifact's own integration test: a chain of three edges with scores 0.9, 0.8 and 0.7, each of which is multiplied by the depth decay $\gamma = 0.8$. Equation 2 gives

$$p \;=\; 0.8^3 \cdot 0.9 \cdot 0.8 \cdot 0.7 \;=\; 0.258048, \tag{6}$$

which is the value the test asserts to a tolerance of 10^{-5}, and the value Figure 2 prints. Applying the verifier with $\lambda = 0.7$ and $v = 1$ gives $c' = 0.258048^{0.7} \approx 0.3874$. The same test file also pins the cross-scope gate, the cycle pruning, the two abstention codes and the calibration's softening behaviour.

Figure 2 shows the worked path together with the three stages of the algebra.

Algorithm 3 The insert trigger that feeds the learned index.

Require: A row (t, r, row) inserted into table t with row id r

for each network N registered with the store **do**
  if N.header.sourceTable $\neq t$ **then continue**
  cols $\leftarrow$ Columns(t) ▷ cached column list
  $j \leftarrow$ index of N.header.textColumn in cols
  if j is undefined **then continue**
  $x \leftarrow \text{row}[j]$ ▷ the text cell
  if $(t, r) \in \text{indexed}[t]$ **then continue**
  $\text{indexed}[t] \leftarrow \text{indexed}[t] \cup \{r\}$
  EnqueueDocument$(N.\text{nlp}, r, x)$
end for

3.3 Coupling a relational store to a learned index

The third seam is between the store and the text pipeline. The store is a page-based key-value engine: 8 KiB pages with a 32-byte header, a write-ahead log with crash recovery, and an encrypted container whose superblock carries a magic number, a format version, the page size, the key-derivation parameters, a salt, a nonce and an epoch, protected by a CRC32. Page contents are sealed with AEAD (AES-256-GCM); the key-encryption key is derived from a passphrase with Argon2id at deliberately expensive parameters, and the data-encryption key is derived from that key and a nonce rather than being stored.

Above the store sits a SQL dialect — not MySQL wire-compatible, but with the same shape: CREATE, INSERT, SELECT, TRAIN, ALTER, INFER, ASSOCIATE, REASON, REASON PATHS, LEARN, CONFIRM, REJECT, CALIBRATE, EXPLAIN and SHOW, plus a set of `mb_*` system tables. The coupling lives in the insert path, and it is given in Algorithm 3.

The idempotence check is the point of Algorithm 3: the set $\text{indexed}[t]$ of already-processed row ids makes the trigger idempotent per (table, row id), so a row cannot be counted twice in the document-frequency statistics, and re-inserting a row after a restart does not double-index it. The trigger runs for every registered network whose header names the table; for tables that no network references, it is a no-op by construction.

The document that is enqueued is put through a pipeline in MIGHTBE-NLP: Unicode normalisation and control-character removal; tokenisation, in which Latin runs become words while CJK characters yield both a unigram and a bigram; vocabulary fitting; document-frequency counting; keyword scoring by TF-IDF Robertson and Zaragoza (2009) and by TextRank, which is a weighted PageRank over vocabulary nodes Mihalcea and Tarau (2004); Page et al. (1999); sliding-window co-occurrence counting; and finally pointwise mutual information

$$\text{pmi}(a, b) \;=\; \ln \frac{P(a, b)}{P(a)\,P(b)}, \tag{7}$$

computed with a denominator floor of 10^{-12}. The resulting association edges are sorted by PMI and written to the `mb_assoc` table; the keyword scores are written to `mb_keywords`. Both are then loaded into the reasoning

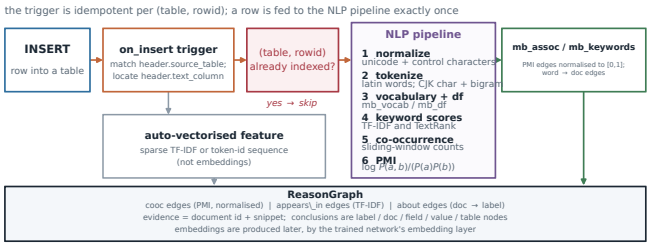


Figure 3: The insert-trigger chain. A row is routed to the text pipeline at most once per (table, row id), and the pipeline's outputs populate the association tables from which the reasoning graph is built.

graph, with PMI min-max normalised over the corpus so that the edge scores lie in $[0, 1]$ and are comparable with the cosine scores described below.

Two clarifications are important and are stated here because they are easy to get wrong from the outside. First, the automatic vectorisation performed at insert time produces a *sparse TF-IDF vector* or a sequence of token ids — not a dense embedding. Embeddings appear only later, when the trained network's embedding layer is queried for a word vector. Second, there is no approximate nearest-neighbour index in this system: the co-occurrence association is a linear filter over a PMI-sorted vector, and the embedding association is a brute-force cosine scan over the vocabulary. The project's own documentation justifies the brute-force scan on the grounds that a vocabulary of at most 20 000 makes it sufficient, and lists an IVF index as future work. We repeat this rather than implying an index that does not exist.

When both kinds of association are available they are fused linearly, $\alpha \cdot \mathrm{emb} + (1 - \alpha) \cdot \mathrm{cooc}$, with a default $\alpha = 0.7$. The two components are put on a common scale deliberately: cosine similarity is mapped affinely from $[-1, 1]$ to $[0, 1]$, and PMI is min-max normalised over the corpus distribution.

Figure 3 shows the whole chain.

4 Implementation

MightBe is a Cargo workspace of nine crates. Figure 4 shows them and their dependency direction.

The counting script in `scripts/count_structure.py` produces the following structural facts directly from the source tree, so that a reader can check them without trusting the project's own documentation:

The crates are `mightbe-core` (tensors, the autodiff tape, layers, optimisers and the plugin host), `mightbe-store` (the paged store, the write-ahead log and the encrypted container), `mightbe-nlp` (the text pipeline), `mightbe-net` (the network registry, the specification builder and checkpoints), `mightbe-reason` (the graph, the search, the calibration and the relation learning), `mightbe-sql` (the SQL dialect and the machine-learning runtime), `mightbe-server` and `mightbe-cli` (a line-oriented

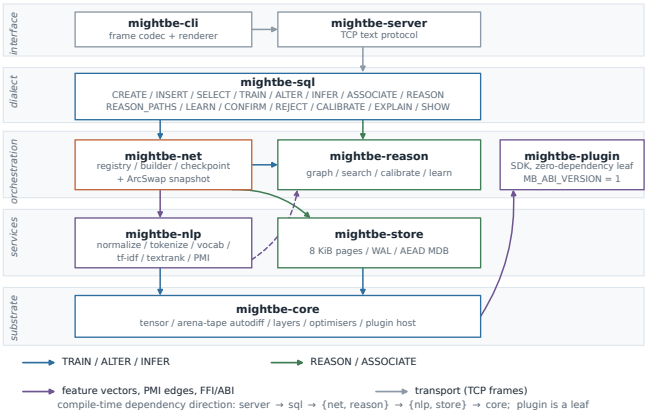


Figure 4: The nine crates of the MightBe workspace and the principal data flows between them.

Table 1: Structural facts counted directly from the source tree by `scripts/count_structure.py`, rather than taken from the project's documentation.

Crates	9
Production source files (9 crates, `src/`)	55
Production source lines	20 353
Integration-test source files	9
Integration-test source lines	3 003
Plugin fixture crates (`tests/`)	3 files, 299 lines
`#[test]` attributes in `src/`	183
`#[test]` attributes in `tests/`	68
Variants of the autodiff tape's `Op` enum	31

TCP protocol and its renderer), and `mightbe-plugin` (the plugin SDK).

Two implementation choices deserve comment because they bear directly on the contributions.

Plugin ABI. The plugin path lets a layer be supplied by a separately compiled shared object. The SDK pins an ABI version, `MB_ABI_VERSION = 1`, and exports ten symbols: `mb_api_version`, `mb_layer_count`, `mb_describe`, `mb_create`, `mb_param`, `mb_infer_shapes`, `mb_forward`, `mb_backward`, `mb_drop` and `mb_last_error`. The host checks the ABI version before loading, and the plugin's backward rule is installed into the tape as an `Op::Plugin` node holding a trait object. Plugins are not sandboxed; the project's non-goals say so explicitly, and we repeat it here because an unsandboxed plugin is a security boundary that does not exist.

Cell DSL. The specification format includes a small expression language used to describe composite, DAG-shaped layers whose sub-computations depend on earlier sub-computations within the same layer. It is deliberately not Turing-complete; the non-goals list states this as a design constraint rather than

Table 2: Outcome of `cargo test --workspace`, parsed from the run log by the summariser.

Test targets	32
unit targets	14
integration targets	9
doc-test targets	9
Tests passed	251
in unit targets	183
in integration targets	68
Tests failed	0
Tests ignored	1 (one `mightbe_plugin` doc-test)
Cargo exit status	0
Slowest target	`tests/mdb.rs`, 78.39 s

a limitation.

5 Evaluation

5.1 What we ran

The repository contains no benchmarks, no performance harness, no accuracy dataset and no latency measurement. The only quantitative result we can honestly report is the outcome of the project's own test suite. We ran the full workspace suite,

```
$ cargo test --workspace
```

and parsed the log.[1] The result was:

The slowest target is slow for a reason worth stating: the container integration test exercises the key-derivation parameters, which are set to be deliberately expensive (Argon2id at $m = 19 \cdot 1024$ KiB, $t = 2$, $p = 1$), so the test's wall-clock cost is dominated by key derivation rather than by I/O.

The per-target breakdown, which is the evidence behind the table above, is:

Three properties of this result should be stated next to it, because a reader is entitled to know what a green suite does and does not establish.

- **The suite is not independent evidence of correctness.** The project's own testing document states that the tests were generated with the assistance of a large language model. We regard this as material: a suite written alongside the implementation can encode the implementation's assumptions, including its incorrect ones. We therefore treat the suite as evidence that the artifact builds and that its invariants hold under its own tests — not as evidence that the artifact is correct.
- **The suite contains no performance assertions.** There is no timing threshold anywhere in it. The one number in Table 2 that is a duration, the 78.39 s of the `mdb` target, is a wall-clock observation from the log, not a bound.
- **The suite exercises the failure paths, not just the happy ones.** The tests named above include the smoke-failure rollback, the abstention codes, the cycle pruning, the cross-scope gate, the plugin panic-isolation case (which deliberately triggers a panic inside a plugin and asserts that the host survives it), and the wrong-ABI and missing-symbol plugin fixtures.

[1] The parser is `scripts/summarise_test_log.py`.

5.2 What we did not run: a protocol, not a result

The three contributions each suggest a measurement that the artifact does not yet support. We state the protocol rather than inventing the numbers. Each item names the script that would produce the value, the metric, and the hypothesis, so that a future version of the artifact can fill the table in without re-deriving the design.

P1: hot-mutation stall. P1 has not been measured, because the script that would produce it, `scripts/bench_alter_stall.rs`, does not exist yet. *Question.* How long does a running network continue to serve the old version while an alteration is in flight, and how does that latency vary with the size of the network and the number of concurrent readers? *Protocol.* A driver issues a steady stream of inference requests against a network while a second thread performs an ALTER that adds a layer, with a read-side snapshot read on every request. Record, per request, the wall-clock latency, the version reported by the snapshot, and whether the request observed the old or the new version. Report the distribution of the latency of requests that observed the old version during the alteration window, and the maximum observed read stall. *Script.* `scripts/bench_alter_stall.rs`, to be added. *Expected shape.* Because publication is a single atomic store, the stall should be independent of the number of readers; we do not know whether that is true until it is measured, and we do not assert it.

P2: migration fidelity. P2 has not been measured, because the script that would produce it, `scripts/bench_migration.rs`, does not exist yet. *Question.* After an alteration that adds or removes a layer, what fraction of the surviving parameters retain their exact pre-alteration values, and which parameters are re-initialised? *Protocol.* For each layer configuration in a swept set, record the parameter set before and after the alteration, the shapes, and a hash of each tensor's contents; report the fraction preserved, the fraction re-initialised, and the set of warnings emitted. The expected outcome, from the code, is that a parameter preserved with an unchanged shape is bit-identical and that a parameter whose shape changed is re-initialised with a warning. *Script.* `scripts/bench_migration.rs`, to be added.

Table 3: Per-target test counts, the evidence behind Table 2. Binary-only targets and doc-tests report zero here.

Target	Tests	Target	Tests
`mightbe_core` (unit)	89	`mightbe_nlp` (unit)	23
`mightbe_reason` (unit)	28	`mightbe_store` (unit)	18
`mightbe_cli` (unit)	13	`mightbe_server` (unit)	7
`mightbe_sql` (unit)	5	`mightbe_net` (unit)	0
`tests/mdb.rs`	13	`tests/keyword.rs`	12
`tests/net_m4.rs`	12	`tests/reason_m6.rs`	8
`tests/ml_e2e.rs`	6	`tests/injection.rs`	5
`tests/plugin_chain.rs`	4	`tests/plugin_hooks.rs`	4
`tests/train_m1.rs`	4	`mightbe_plugin`, CLI and fixture binaries	0

P3: reasoning quality. P3 has not been measured, because the script that would produce it, `scripts/bench_reasoning.rs`, does not exist yet. *Question.* How well does the confidence of Equation 2 and Equation 3 agree with the correctness of the answer? *Protocol.* On a labelled question set, record for each answer its confidence, whether the answer was correct, and the size of its evidence set. Report the expected calibration error over ten bins before and after the temperature calibration of Equation 5, the area under the precision-recall curve as a function of the abstention threshold, and the selective-risk curve. *This requires labelled data, which the repository does not contain*; the corpus, the question set and the labelling protocol would all have to be built, and until they are, no accuracy or calibration number can be reported. *Script.* `scripts/bench_reasoning.rs`, to be added.

P4: trigger throughput. P4 has not been measured, because the script that would produce it, `scripts/bench_trigger.rs`, does not exist yet. *Question.* What does the insert trigger cost per row? *Protocol.* Insert rows in batches of fixed size into a table designated by a network's header, with the trigger enabled and disabled, and report the wall-clock cost per row of the trigger alone (by differencing), the number of documents enqueued, and the size of the resulting association table. Note that the pipeline's cost is dominated by a full rebuild of the vocabulary and the co-occurrence statistics, so the interesting quantity is the amortised cost per inserted row, not the latency of a single insert. *Script.* `scripts/bench_trigger.rs`, to be added.

Each item above says explicitly that it has not been measured, so that a reader can see where the gaps are. We do not provide placeholder tables with invented values; an empty table with a protocol is more useful to a reader than a populated table that is fiction.

6 Threats to Validity

The test suite was machine-generated. As noted above, the artifact's own testing document records that its tests were produced with the assistance of a large language model. The suite therefore encodes the authors' (and the generator's) assumptions about what the system should do. It is strong evidence that the code runs and that its internal invariants hold under its own tests; it is weak evidence that the system does the right thing. A reader should weight Section 5.1 accordingly.

One path score is pinned; the algebra is not validated. The score of Equation 6 is asserted by an integration test to a tolerance of 10^{-5}, which makes the specification checkable at one point. It says nothing about whether the constants — the depth decay, the cross-scope gate, the verifier exponent — are sensible, because no labelled data exists against which to check them.

The corpus is synthetic. The repository's example data is small and machine-generated. Any statement in this paper about behaviour on realistic text is therefore a statement about the code path, not about a workload.

Concurrency is tested, not measured. The snapshot tests assert atomicity of the publication mechanism. They do not establish that the system is free of data races under load, and no sanitiser run or stress test is reported here.

Scope of the claim. We claim a coherent design and a checked implementation of three specific mechanisms. We do not claim competitive performance, we do not claim that the confidence is well calibrated, and we do not claim that the system is suitable for production deployment.

7 Related Work

Automatic differentiation engines. MightBe's tape follows the standard reverse-mode arrangement described by Rumelhart, Hinton and Williams Rumelhart et al. (1986): an explicit computation graph with per-node gradient accumulation. Its distinguishing feature relative to the widely used frameworks is not the differentiation but the mutation protocol: PyTorch Paszke et al. (2019) and TensorFlow Abadi et al. (2016) both support graph mutation, but they expose it as a

Python-level operation whose failure modes are not, in general, atomic with respect to concurrent readers, and neither provides a mechanism that verifies a mutated graph before it is published. MightBe's contribution is the combination of a validation gate, a smoke gate, and an atomic publication step, not a new differentiation algorithm.

Optimisers. Adam Kingma and Ba (2014), SGD with momentum Rumelhart et al. (1986), and gradient clipping in the form of Pascanu et al. Pascanu et al. (2013) are used as published. We contribute nothing to optimisation; the relevant design decision is that the optimiser state is keyed by parameter name so that it survives structural change.

Probabilistic ranking and term weighting. The keyword scorer follows the TF-IDF family as formalised in the probabilistic ranking framework of Robertson and Zaragoza Robertson and Zaragoza (2009) and its ancestors Robertson and Walker (1994); TextRank Mihalcea and Tarau (2004) applies a PageRank-style stationary distribution to a graph of words, in the line of Page et al. Page et al. (1999). MightBe's use of these is straightforward and unmodified. Its own contribution here is the decision to keep the two keyword scores separate in the keyword table, and to combine them with a pointwise-mutual-information association measure computed over the same corpus Page et al. (1999).

Similarity hashing and near-duplicate detection. Charikar's SimHash Charikar (2002) is a canonical technique for approximate near-duplicate detection, and is mentioned in the artifact's documentation as a direction for the future. It is not implemented, and we list it among the planned features rather than among the contributions.

Explainable inference and calibration. Work on explaining model predictions, on producing evidence alongside answers, and on calibrating confidence scores is extensive; the calibration machinery in this artifact follows the standard expectation-calibration-error formulation Guo et al. (2017); Naeini et al. (2015). Our contribution is narrower and compositional: the confidence of an answer is a function of the scores along an explicit path, so that it can be recomputed by hand from the returned path, and the returned evidence is derived from that same path rather than looked up afterwards.

Transactions and relational foundations. MightBe's store is a paged, logged, encrypted key-value engine with a SQL dialect on top, in the tradition of Gray and Reuter's transaction processing Gray and Reuter (1992) and the relational model of Codd Codd (1970). The store is deliberately a small subset: there is no join optimiser, the isolation level is read uncommitted, and the undo log is held in memory. We cite this literature to position the storage layer, not to claim parity.

8 Limitations

This section is specific by design; the list below is drawn from the artifact's own documentation and from the source tree.

No benchmarks, no accuracy numbers, no latency numbers. The repository contains no performance harness and no labelled evaluation set. Nothing in this paper reports a speed or an accuracy, and Section 5.2 specifies what would be needed to report one.

The test suite is machine-generated. The project's testing document states that the tests were generated by a large language model. This is disclosed in the abstract and again in Section 5.1 because it materially changes how the suite should be read.

Six named modules do not exist. The project's documentation lists `reason/verifier.rs`, `reason/evidence.rs`, `schema/profile.rs`, `schema/blocking.rs`, `schema/features.rs` and `schema/learn.rs` as planned. None of these files is present in the source tree. We therefore describe the verifier, the evidence policy and the schema-learning facilities through their interfaces and their behaviour inside the modules that do exist, and we do not list them as contributions.

No approximate nearest-neighbour index. Retrieval is a linear filter over a PMI-sorted vector plus a brute-force cosine scan over the vocabulary. There is no IVF index, no HNSW graph, and no SIMD path. The project's own documentation justifies the brute-force scan for vocabularies of at most 20 000 and lists an IVF index as future work; we make no claim about scalability beyond that.

Not a production database. There is no join optimiser (joins are nested loops with a primary-key filter), no window functions, no stored procedures, no user-defined functions in Python, the isolation level is read uncommitted, and the undo log is in memory. The wire protocol is a text frame format of our own design; it is not MySQL wire-compatible, and no such compatibility is claimed.

No GPU acceleration and no distributed training. Both are explicit non-goals. All computation is on the CPU, in a single process.

The tokenizer is not a high-precision multilingual tokenizer. CJK text is segmented into characters and character bigrams, which is a deliberate simplification; the project's documentation states that no precision guarantee is offered for multilingual text.

Reasoning depth is hard-capped. The beam search explores at most three hops by default, with a beam width of 16 and a branch factor of 8. This is a bound, not a heuristic; deeper answers are simply not reachable, and the bound is a configuration value rather than an adaptive one.

Learned relations are candidates, not facts. Relations discovered by LEARN are evidence-bearing candidates that a user must CONFIRM; REJECT discards them. Nothing in the system promotes a learned relation to a fact automatically.

The reasoning layer does not generate text. It returns nodes, confidences, paths and evidence. Any natural-language rendering is the client's responsibility, and none is provided.

Plugins are not sandboxed. A plugin is an ordinary shared object loaded into the host process. The ABI version is checked, and the host survives a panic inside a plugin, but a plugin can do anything the host can do.

The Cell DSL is not Turing-complete. This is a design constraint rather than an unfinished feature, but it does bound the class of layers that can be expressed declaratively.

The security model is local listening. The server binds locally and relies on the local trust boundary. There is no TLS, no authentication and no multi-tenancy.

The project is framed as a blueprint. The repository's own documentation describes itself as a blueprint for a database with built-in machine learning, and its milestone table marks several capabilities as partially implemented or not started. We have read the code rather than the documentation wherever the two disagreed, and several of the discrepancies we found are noted above.

9 Conclusion

We have described a single-process system in which a neural runtime can be structurally mutated while it serves, a reasoning layer can answer with a path and an evidence set whose confidence can be recomputed by hand, and a relational store feeds a learned index through an idempotent trigger. The three mechanisms share a design attitude: every state transition that a reader can observe is either atomic or explicitly refuses, and every number that is reported can be traced to a rule or to a file.

The contribution we can defend today is the specification and its implementation, not a set of measurements. The test suite passes — 32 targets, 251 tests, one ignored doc-test — and the specification's one worked example is pinned by an integration test to a tolerance of 10^{-5}. The measurements that would tell a reader whether any of this pays off in practice do not exist yet, and Section 5.2 states what they would take.

References

M. Abadi, P. Barham, J. Chen, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.

M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 380–388, 2002.

E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1321–1330, 2017.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014.

R. Mihalcea and P. Tarau. TextRank: Bringing order into text. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 404–411, 2004.

M. P. Naeini, G. F. Cooper, and M. Hauskrecht. Obtaining well calibrated probabilities using Bayesian binning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 2901–2907, 2015.

L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1310–1318, 2013.

A. Paszke, S. Gross, F. Massa, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, 2019.

S. Robertson and H. Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009.

S. E. Robertson and S. Walker. Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 232–241, 1994.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323 (6088):533–536, 1986.