Sutra: Tensor-Op RNNs as a Compilation Target for Vector Symbolic Architectures

Anonymous Author(s)
Affiliation
Address
email

Abstract

Sutra is a typed, purely functional programming language whose compiled forward pass is a PyTorch neural network. The compiler beta-reduces the whole program (primitives, control flow, string I/O) to a fused tensor-op graph: rotation binding, unbind, bundle, polynomial Kleene three-valued logic, and tail-recursive loops all lower to tensor operations on a frozen embedding substrate, with the only remaining host-side control flow a thin tick-loop that breaks when a halt scalar saturates. The Kleene connectives are Lagrange-interpolated polynomials exact on the $\{-1, 0, +1\}$ truth grid; rotation binding doubles as the language's hash-map primitive (Haar-orthogonal role rotations seeded by content hash). The substrate is the architecture target: swap the embedding model and the same source recompiles against a different geometry.

The validation is a single fact testable two ways. (1) The same program runs on four frozen embedding substrates spanning two modalities (three text encoders: nomic-embed-text, all-minilm, mxbai-embed-large, and one protein language model: ESM-2) and decodes bundles at 100% accuracy through width k=8 on every one, where the textbook Hadamard product has already collapsed (2.5% on mxbai-embed-large, 7.5% on all-minilm); single-cycle bind/unbind round-trips at $\approx 1.5\times10^{-15}$. A Sutra program's inputs and outputs are embeddings in the substrate's vector space; a compile-time codebook handles string literals at the source level and nearest-string lookup at the output boundary. (2) PyTorch autograd flows through the compiled graph end-to-end: a symbolic if-then program of fuzzy rules over 20 classes / 992 words, with a rule tree nineteen ANDs deep, trains from random init (4%; chance = 5%) to 95% within 50 epochs and holds through 300 without any modification to the symbolic source. Gradient descent moves the embeddings the rules evaluate against, leaving the rule graph itself untouched.

This collapses the boundary between writing a logic program and training a neural network: one artifact, two interpretations.

1 Introduction

A frozen embedding model maps strings (or amino-acid sequences, or any other input the model was trained on) into a deterministic continuous vector space. Given such a substrate, two technical questions follow:

1. **Which operations on these embeddings are reliable enough to be used as primitives** of a compositional algebra over the substrate's vector space?
2. **What is the correct binding operation?** Hyperdimensional computing's textbook bind operators (Hadamard product, circular convolution) were derived assuming hypervectors

Submitted to 40th Conference on Neural Information Processing Systems (NeurIPS 2026). Do not distribute.

drawn from a controlled random distribution. Frozen LLM embeddings are not such a distribution. §3.2 measures four substrates and reports that rotation binding decodes at 100% accuracy through bundle widths where Hadamard has already collapsed.

This paper answers both questions in the form of a working programming language, **Sutra**, whose primitives are these consolidated operations and whose compiled forward pass is a PyTorch neural network. The naming: **Sutra** is the Sanskrit *sūtra* (thread, rule, aphorism), the term for Pāṇini's foundational Sanskrit grammar.

1.1 Contributions

The four core technical contributions of this paper are:

1. **Polynomial fuzzy logic via Lagrange interpolation of Kleene's three-valued truth tables.** The truth axis encodes $T = +1$, $U = 0$, $F = -1$. On the discrete $\{-1, 0, +1\}$ grid, the Kleene connectives are $\mathrm{AND} = \min$, $\mathrm{OR} = \max$, $\mathrm{NOT} = -\cdot$. The min/max forms (the standard Gödel t-norm/t-conorm choice; Hájek 1998) are non-differentiable at the diagonal $a = b$, which breaks gradient flow when connectives compose with the tensor-op graph (van Krieken, Acar & van Harmelen 2022 survey the issue across t-norm-derived neural-symbolic operators). Sutra resolves this by Lagrange-interpolating each connective as a polynomial that is exact on the 3×3 Kleene grid and C^∞ elsewhere:

$$\begin{aligned} \mathrm{AND}(a,b) &= \tfrac{1}{2}(a + b + ab - a^2 - b^2 + a^2b^2) \\ \mathrm{OR}(a,b) &= \tfrac{1}{2}(a + b - ab + a^2 + b^2 - a^2b^2) \\ \mathrm{NOT}(a) &= -a \\ \mathrm{XOR}(a,b) &= -ab, \qquad \mathrm{XNOR}(a,b) = ab \end{aligned}$$

   {AND, OR, NOT} is functionally complete for the Kleene fragment; XOR/XNOR collapse to a single multiplicative term because their interpolant is zero whenever either input is U and bilinear in the {−1, +1} corners. Every Kleene-valid connective is therefore a polynomial tensor-op-graph fragment, gradient-compatible, branchless, and exact on the discrete-logic regime. A symbolic if-then rule built from these gates is one fused subgraph that PyTorch autograd backprops through end-to-end (§3.6).

2. **Beta reduction to a substrate-pure tensor-op graph.** The compiler inlines stdlib operator definitions, beta-reduces through bound names, then runs an algebraic-simplification pass over the residual. What's left is a fused tensor-op graph (matmul / element-wise / nonlinear) with no named bindings or function calls. Three concrete moves go beyond standard inlining + constant folding: conditionals lower to soft-mux polynomials ($\frac{1+\mathrm{cond}}{2}\,a + \frac{1-\mathrm{cond}}{2}\,b$) so the compiled artifact has no `if` opcodes; Haar-orthogonal binding rotations `R_role` are materialized at compile time so runtime `bind` is one matmul against a constant matrix; canonical synthetic axes are assigned compile-time so every primitive-type read/write is a known index, not a hashtable lookup. §4.2 traces this lowering stage-by-stage on a concrete program; the compilation pipeline as a whole is diagrammed in Appendix J.

3. **Tail recursion as the loop primitive.** Loops are tail-recursive function declarations (`do_while`, `while_loop`, `iterative_loop`, `foreach_loop`) whose body's `return NAME(args)` becomes the recurrent step. Each loop compiles to a soft-halt RNN cell with substrate-pure halt detection (heaviside → cumulative monotone halt → soft-mux state freeze). The body of every loop tick is one straight-line tensor pipeline with no in-graph branches; a thin Python `while True: ... break` driver wraps the body and terminates when the halt scalar saturates (§3.4). The state vector is fixed-width across iterations, **O(1) state, O(N) compute, O(N) gradient tape during training**, where N is iterations actually executed.

4. **Synthetic-dimension rotation binding as an angular hash map.** The compiler reserves a synthetic block of canonical dimensions and uses Haar-orthogonal rotations seeded from the role's content hash to bind keys to slots. To the authors' knowledge this is the first use of a high-dimensional rotation pattern as the substrate for a functional hash-map primitive.

These four primitives integrate into a single working compiler that lowers `.su` source to a self-contained PyTorch module on CPU or CUDA. Program inputs and outputs are embeddings in the substrate's vector space; a compile-time codebook (implemented with an embedded vector database, §3.5) handles the convenience of source-level string literals and nearest-string output.

1.2 The substrate is the architecture target

A Sutra program is compiled for an *embedding-space architecture*, the way a C program is compiled for x86 and a CUDA kernel for an NVIDIA SM. The embedding model fixes dimensionality, the geometry of the semantic block, and the meaning of every basis-vector lookup; swap the model and the same source recompiles to a different `.sdb` codebook against a different geometry. The substrate need not be an LLM, it can be any network producing a dense vector representation, including the hidden state of a trained model. §3.2's ESM-2 protein-LM row demonstrates this substrate-agnostically.

---

2 Related Work

2.1 Vector Symbolic Architectures

VSA is a family of algebraic frameworks for computing with high- dimensional vectors (Kanerva 2009; Plate 1995; Gayler 2003). The standard VSA development assumes hypervectors drawn from a controlled random distribution designed for the algebra; bind is typically Hadamard product or circular convolution. Frozen LLM embedding spaces are not designed for VSA, and the textbook bind operations do not always transfer cleanly to them. Rotation binding (`R_role @ filler` for a role-seeded Haar-random orthogonal `R_role`) is the choice that worked across the substrates we tested, and is what Sutra uses today; §3.2 reports the per-substrate measurements supporting that choice.

The closest software peer in the VSA space is **TorchHD** (Heddes et al. 2023), a PyTorch library that exposes VSA primitives (bind, bundle, similarity) as tensor operations. Sutra and TorchHD differ on what the user writes and what the compiler does:

- **TorchHD is a *library*.** The user writes Python code that calls TorchHD primitives; control flow is host-side Python; there is no source-language layer above the primitives, no compile step, and no algebraic reduction across primitive calls. Each primitive call is a tensor op, but the program itself is a Python function with whatever control flow the user wrote.
- **Sutra is a *language with a compiler*.** The user writes `.su` source which the compiler beta-reduces to a substrate-pure tensor-op graph (§1.1-2): a single straight-line graph of matmul / element-wise / nonlinear ops with no Python control flow. Loops are tail-recursive function declarations that lower to soft-halt RNN cells; conditionals are differentiable fuzzy interpolations rather than Python `if`. Hash-map structure is implemented via synthetic-dimension rotation, not via a host-side dictionary.

A second axis where Sutra differs from existing HDC software is **string I/O**. TorchHD and similar libraries expose the algebra over user-supplied hypervectors; the user maintains a `dict[str, hypervector]` and an explicit codebook tensor by hand. Sutra's compile-time codebook (§3.5) closes that loop: every embedded string in `.su` source is embedded once at compile time via the configured frozen LLM, stored in the project's `.sdb` codebook, and decoded at the program output via `nearest_string`. The frozen-LLM embedding is load-bearing, random hypervectors yield a working VSA algebra with no I/O story.

The structural differences (Sutra contains no Python, the string-to-vector map and codebook are constructed by the compiler rather than by the user, and the whole program reduces to a single fused tensor-op graph) are differences in artifact shape, not library speed.

2.2 Comparison to other neuro-symbolic languages

The closest neuro-symbolic-language peers are **Scallop** (Li et al. 2023, Datalog with provenance-semiring differentiability), **DeepProbLog** (Manhaeve et al. 2018, ProbLog with neural predicates), **Logic Tensor Networks** (Badreddine et al. 2022, first-order logic compiled to t-norm losses), and **NeurASP** (Yang et al. 2020, Answer Set Programming with neural predicates). All share a two-stage perception-then-reasoning shape: a neural model extracts discrete symbols from raw input, and a symbolic program reasons over those symbols. Sutra's shape is different at this architectural level: the substrate is a continuous embedding space throughout, primitives operate on vectors end-to-end, and the whole program (including what would be the logic program in Scallop) compiles to a single fused tensor-op graph through beta reduction. There is no discrete symbolic stratum to extract into or reason over; differentiability is inherited from the tensor-op graph itself, not from a provenance annotation on a relational query. The two are good at different problem structures: Scallop and its peers when the problem is naturally relational and perception cleanly factors out; Sutra when computation is best expressed as algebra on vectors over a substrate the program reads strings into and decodes strings out of.

The closest HDC peer with compiler infrastructure is **HDCC** (Vergés et al. 2023), a description-file DSL targeting self-contained C for embedded classification, random/level hypervectors only, no general control flow, scoped to classification. **TorchHD** and OpenHD / HDTorch are libraries without a language-level loop primitive. To the authors' knowledge, no published HDC system combines (a) one fused tensor-op graph as compile target, (b) HDC primitives as the operations, (c) a frozen externally-trained vector embedding space as the substrate, and (d) tail-recursive loops compiled to soft-halt RNN cells with constant state-vector width in recursion depth. The combination is what distinguishes Sutra, not any one of those properties in isolation.

2.3 Differentiable Programming, AOT Compilation, and Knowledge

Compilation

The closest design ancestors are partial-evaluation systems that specialize programs at compile time (the Futamura projections), differentiable programming systems that treat programs as differentiable functions (JAX), AOT compilation of neural networks (TVM, XLA), and knowledge compilation in symbolic AI (Darwiche & Marquis 2002). Sutra differs from each: TVM/XLA start from a network, not toward one; JAX treats programs as differentiable but does not bake source literals into weights; partial evaluation specializes for compile-time-known values but does not target a neural-network-shaped artifact; knowledge compilation targets Boolean circuits, not continuous embedding spaces. Sutra's combination (fold source literals into the weight structure, compile control flow to RNN cells, run the whole program as one tensor-op graph over a *continuous* substrate) is the novel position.

---

3 Consolidation into Canonical Primitives

The central design move: hold the operation interface fixed and pick a binding implementation that works on dense externally-trained substrates. Standard VSA's Hadamard product fails here, elementwise multiplication of correlated real-valued vectors produces destructive crosstalk on bundled retrieval (§3.2 measures this directly). Rotation binding works: each role gets a Haar-random orthogonal `R_role` seeded by `hash(role)`, and `bind(role, filler) = R_role @ filler` is invertible (unbind is the transpose) and well-conditioned. The compiler caches `R_role` per-role at module init so runtime bind is a single matmul against a precomputed matrix.

3.1 Notation

We work in $\mathbb{R}^d$ with d the substrate's embedding dimension (768 for nomic-embed-text). Every value has the layout $[\,\text{semantic} \mid \text{synthetic}\,]$. The seven primitive operations: $\text{bind}(r, f) = R_r f$ where $R_r = \text{QR}(\text{hash}(r)).Q$ is Haar-orthogonal, $\text{unbind}(r, v) = R_r^\top v$, $\text{bundle}(x, y) = (x + y)/(\|x + y\| + \varepsilon)$, $\text{similarity}(x, y) = (x \cdot y)/(\|x\|\,\|y\| + \varepsilon)$, $\text{normalize}(v) = v/(\|v\| + \varepsilon)$, the Lagrange Kleene gates as in §1.1-1, and the soft-halt cell of §3.4. Full signature/definition table and the soft-halt cell update equations are in Appendix A.

3.2 Capacity of rotation versus Hadamard binding across substrates

We measure decode accuracy as a function of bundle width k on real embeddings across four substrates spanning two modalities: three frozen LLM text encoders (nomic-embed-text, all-minilm, mxbai-embed-large) and one frozen protein language model (ESM-2 small, `facebook/esm2_t6_8M_UR50D`). LLM substrates embed an 84-word noun vocabulary; the ESM-2 substrate embeds an 84-sequence amino-acid vocabulary (full protocol in Appendix C). For each bundle width and binding scheme we run 10 trials, sampling k random (role, filler) pairs without replacement, forming the bundle, and decoding by unbind + argmax-cosine against the full codebook. *Rotation binding* uses a role-seeded Haar-orthogonal `R_role`; *Hadamard binding* is the textbook elementwise product (MAP-VSA).

Cross-substrate decode accuracy at representative widths (full k $\in$ {2, 4, 8, 16, 24, 32, 48} sweeps in Appendix C):

substrate (dim)	rotation k=8	rotation k=48	Hadamard k=8	Hadamard k=48
nomic-embed-text (768)	100.0%	93.3%	87.5%	48.3%
all-minilm (384)	100.0%	42.3%	7.5%	1.7%
mxbai-embed-large (1024)	100.0%	72.1%	2.5%	1.0%
ESM-2 (320)	100.0%	44.2%	28.7%	4.2%

ESM-2 (Lin et al., Science 2023) is a protein language model trained on UniRef with no natural-language exposure; the same rotation-vs-Hadamard pattern reproduces in that modality. Rotation reversibility round-trip across all four substrates: mean $\|\text{unbind}(R, \text{bind}(R, x)) - x\| = 1.5 \times 10^{-15}$ (floating-point round-off, Q orthogonal). Reproduction: `experiments/rotation_binding_capacity_{llm,bioinformatics}.py`.

3.2.1 Noise accumulation across chained bind/unbind cycles

The §3.2 protocol measures one bind+bundle+unbind cycle. Nested records (a recovered filler becoming the role of a sub-record) add bundle noise per level. We measured this directly: chain lengths L $\in$ {1, 2, 4, 8, ...}, 20 trials, bundle width 4. Raw accuracy holds at 100% through L=2 on every substrate and falls to chance (1/84) by L=8. The demonstrated regime is therefore single-cycle records, which matches the shape of the `role_filler_record`, `knowledge_graph`, and predicate-lookup demos. Pure rotation chains without per-step distractor bundling remain exact (round-trip 1.5×10^{-15} per cycle), so the noise mechanism here does not apply to the soft-halt loop cell of §3.4. Reproduction script: `experiments/crosstalk_chain.py`; full per-substrate L-sweep tables in Appendix D.

3.3 The extended-state-vector layout

Every value carries a fixed `[semantic | synthetic]` layout: the d-dimensional semantic block holds the substrate embedding for vector-shaped values, and a small synthetic block reserves canonical axes for primitive types (real, imag, truth, char) and a loop-completion flag, with the remaining axes paired into 2D Givens planes for variable slots. Default at d = 768 (nomic-embed-text): a 100-dim synthetic block accommodates the five canonical axes plus 47 disjoint slots. Rotation binding is block-diagonal across the split (`Q_role` is Haar-random in the semantic block, identity on the synthetic block), so the synthetic axes pass through bind/unbind unchanged, a fuzzy-truth scalar can coexist with a semantic vector inside the same value without bind smearing them. Full per-axis purpose table and slot allocator details in Appendix B.

3.4 First-class loops as RNN cells

Runtime data-dependent loops compile to **self-halting RNN cells**. Each tick: snapshot pre-step state, evaluate halt on the substrate (truth-axis read $\rightarrow$ heaviside $\rightarrow$ cumulative saturating sum to `halted`), run the cell body, soft-mux between pre- and new-step state by `halted`. A Python `while True:` driver breaks the moment `halted` saturates; this is the only host-side branch in the loop

machinery. Inside the cell body, every operation is a substrate tensor op. No compile-time iteration cap, programs terminate when their halt condition fires. Standard PyTorch tracing handles a Python while-loop wrapping pure tensor ops; autograd records each iteration as it executes, which is the mechanism §3.6 relies on for backprop through the cell. Figure~1 visualizes one tick.

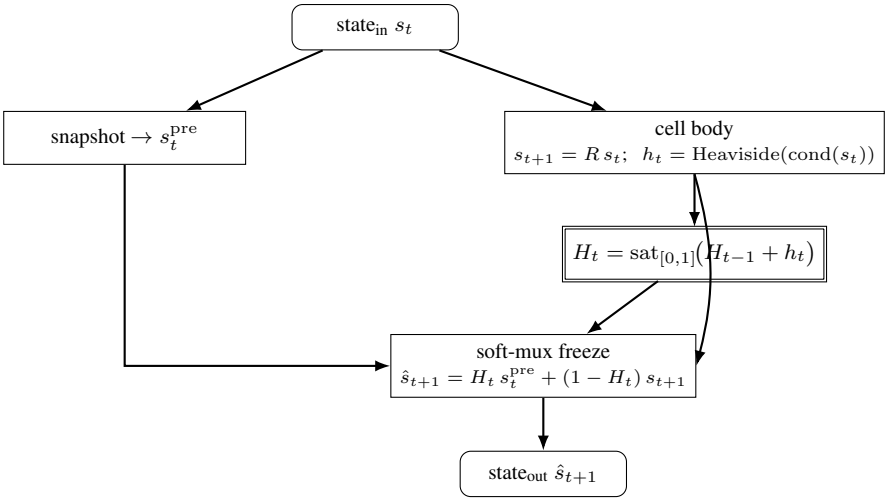

Figure 1: Per-tick dataflow of the soft-halt RNN cell. Once H_t saturates at 1, the soft-mux output equals s_t^{pre}, the loop has frozen. The cumulative halt H_t acts as a boundary read of the same shape as the codebook decode (§3.5).

Constant memory in recursion depth. The state vector is fixed-width and shared across iterations, so a tail-recursive loop consumes O(1) memory in the state vector regardless of trip count. Compute is O(N) and the autograd tape during training is O(N) in iterations actually executed (standard PyTorch, freed after backward). To the authors' knowledge no other HDC system or compiler exposes user-program-level recursion: HDCC is scoped to classification pipelines, TorchHD requires the user to write Python loops over hypervectors. The recurrent shape that emerges is what Siegelmann & Sontag (1992) showed computes any Turing-machine-computable function with rational weights.

3.5 I/O is in the embedding space; the codebook is a comfort layer

A Sutra program's inputs and outputs are embeddings in the substrate's vector space. Strings are a convenience for writing source-level literals: every string literal in `.su` source is embedded once at compile time and stored in a **codebook** (implemented as an embedded vector database with an HNSW index, on disk as a `.sdb` file shipped alongside the compiled module). At the program's output boundary, the runtime decode `_VSA.nearest_string(query)` maps a query embedding to the nearest stored string when the program's caller wants a string back. Calling the codebook at this boundary is shape-equivalent to calling PyTorch for a matmul, neither is the kind of host-side control flow substrate purity forbids. Implementation details (RDF triple layout, HNSW parameters, `.sdb` file format, complexity analysis) are in Appendix E.

3.6 End-to-end differentiable training through Sutra operations

Because every Sutra primitive compiles to a differentiable tensor operation, the compiled graph supports standard PyTorch `loss.backward()` without modification. We verify this by training learnable parameters through a fuzzy-logic classifier built entirely from Sutra operations.

Setup. 992 words across twenty semantic categories (50 each, deduplicated; full list in Appendix G) are embedded via nomic-embed-text (768-d, frozen). Twenty learnable prototype vectors are initialized randomly. The classifier computes cosine similarity between input and each prototype and applies a Lagrange-interpolated fuzzy if-then rule:

$$\text{rule}_i \;=\; \text{AND}\Big(\text{sim}(x, p_i),\; \bigwedge_{j \neq i} \text{NOT}\big(\text{sim}(x, p_j)\big)\Big)$$

with the AND-of-NOTs left-folded across $K - 1$ other classes (so the $K = 20$ rule nests nineteen ANDs deep). Full-batch cross-entropy over the twenty rule scores drives Adam updates (lr=0.005) on the prototype embeddings.

Results. Random init: 4% accuracy (chance = 5%). Training reaches 95% by epoch 50 and holds through epoch 299, loss converging to 1.154. Gradient norms at all twenty prototypes are nonzero throughout (range 0.94–4.20), so backprop reaches every learnable parameter through `similarity` $\rightarrow$ `fuzzy_not` $\rightarrow$ nineteen nested `fuzzy_and` $\rightarrow$ cross-entropy.

Phase	Accuracy	Loss
Before	4%	3.01
After	95%	1.15

Appendix K diagrams the explicit graph for $K = 3$; the $K = 20$ graph used in the experiment has the same shape with twenty learnable prototypes and the AND-of-NOTs left-folded across nineteen $\mathrm{NOT}(\mathrm{sim})$ terms. The input embedding fans out to K cosine-similarity nodes against K learnable prototypes; each `sim_i` enters one branch of an AND-tree (the i-th rule takes `sim_i` directly and `NOT(sim_j)` for j $\neq$ i); the K rule scores are stacked, scaled by temperature, softmaxed, and cross-entropied against the label. Every node is a PyTorch tensor op, no Python branches and no string-keyed lookup, and backprop reaches every learnable parameter through the same compiled graph that runs at inference.

At K=20 the rule for class i is an AND of `sim(x, proto_i)` with a left-folded chain of nineteen `NOT(sim)` terms, a tensor pipeline that could naively saturate or vanish gradients somewhere along the chain. Empirically it doesn't: every prototype receives a nonzero gradient, accuracy reaches 95% on a vocabulary 70$\times$ larger than the K=3 setting (15 $\rightarrow$ 992 words), and the symbolic program text is unchanged across training. The remaining 5% gap reflects irreducible semantic overlap (e.g. *salmon* fits food and color); gradient norms remain bounded above zero throughout, so this is the optimizer plateauing under those overlaps, not gradient pathology. Standard `torch.autograd` suffices (no Sutra-specific autograd machinery) because the compiler emits only operations PyTorch already knows how to differentiate. Reproduction: `experiments/differentiable_training.py` + raw JSON.

3.7 Type system and surface syntax

Sutra's surface syntax is typed: every value carries a primitive class from a fixed set (`int`, `float`, `complex`, `char`, `bool`, `fuzzy`, `trit`, `vector`, `matrix`, `permutation`, `map`, `string`, `scalar`, `void`), and the type drives the synthetic-axis allocation in the extended layout (Appendix B). Type information is pre-compile-time annotation in the TypeScript sense: it is read by the inliner and the layout pass before the tensor graph is built, but it is opinionated rather than authoritarian. A divergent assignment warns and still emits a graph, because the runtime guarantee is mathematical not structural; a type mismatch produces a semantically meaningless but mathematically valid output rather than a runtime exception. The surface itself presents `if` / `while` / `for` / assignment forms that read imperatively for ergonomic familiarity; the inliner and egglog simplifier (§4) beta-reduce these into the functional tensor-op core, so the imperative-looking source is a veneer over the same compiled graph a hand-written functional spec would produce.

---

4 The Sutra Compiler

The compiler is a five-stage pipeline:

1. **Lex + parse**: `.su` source $\rightarrow$ AST.
2. **Inline + simplify**: stdlib operator definitions inlined; an egglog-based simplifier folds equivalent expressions and runs common-subexpression elimination over the algebra.
3. **Codegen**: AST $\rightarrow$ Python source emitting PyTorch tensor ops. The emitted module includes the runtime class (`_TorchVSA`) as inline source so the artifact is self-contained.

4. **Compile-time substrate population**: embed_batch fetches embeddings for every string literal; `populate_sutradb` pushes the codebook into SutraDB; `prewarm_rotation_cache` precomputes role rotations.
5. **Execute**: emitted module loaded; chosen device (CUDA or CPU) initialized at module import; `main()` called; result returned.

The runtime class is emitted inline rather than imported because the emitted module *is* the substrate-pure tensor-op graph; every compile-time decision (extended-state-vector dimensions, codebook contents, role rotations, SutraDB path, optional `torch.compile`) is baked into the emitted source. Stages 1–4 run at compile time and stage 5 is the runtime forward pass; Appendix J diagrams the pipeline as a vertical flow with the residual at each stage.

4.1 Substrate-purity invariants

Three invariants the compiler enforces: (1) every primitive runs on the substrate (numpy is allowed only at compile time for codebook construction and rotation pre-warm, never on the runtime hot path); (2) no scalar extraction inside an operation: operations may not unpack a Python float from a substrate vector, do scalar arithmetic, and pack the result back; (3) no Python control flow inside an operation: loop halt uses substrate primitives (`heaviside`, `saturate_unit`) instead of Python ternaries.

4.2 Compile-time resolution of role rotations

The central compile-time mechanism that lets the compiler emit a substrate-pure tensor-op graph is **precomputed rotation matrices**: every role rotation is constructed at compile time (`prewarm_rotation_cache`) and stored as a constant tensor. At runtime, `bind(role, filler)` is a single matmul against a precomputed matrix, the compile-time resolution eliminates the QR construction from the runtime graph entirely. Role rotations are runtime constants, like neural-network weights at inference; opt-in `torch.compile` (`SUTRA_TORCH_COMPILE=1`) further folds the per-tick loop body into a single fused kernel. Appendix F traces the lowering of `encode2(r_a, f_a, r_b, f_b) := bundle(bind(r_a, f_a), bind(r_b, f_b))` through every reduction stage; the bottom of the chain contains no `bind`/`bundle`/ `normalize` symbol and no Python control flow, so surface lambda calculus and runtime tensor arithmetic are two notations for the same computation.

---

5 Demonstration, limitations, and future work

The smoke test (`examples/_smoke_test.py`) runs 10 demonstration programs end-to-end across 27 `.su` files (Appendix I); loop coverage lives in `examples/do_while_adder.su` and the 23-case `test_loop_function_decl.py` suite, and the §3.6 differentiable- training experiment uses the same primitive set. The embedded codebook covers the compile-time embed $\rightarrow$ runtime decode path; extended features (hashmap routing, persistent codebook via `SUTRA_DB_PATH`) are deferred pending a concrete requirement.

---

6 Conclusion

Sutra compiles a typed pure-functional source language to a substrate-pure PyTorch tensor-op graph: one vector layout per value, one tensor op per primitive, one dataflow graph per program, no type dispatch at the leaves. With the language in hand, asking which embedding operations compose at what capacity on which substrates becomes a program to write.

References

- Darwiche, A., & Marquis, P. (2002). A knowledge compilation map. *JAIR* 17:229–264.
- Gayler, R. W. (2003). Vector symbolic architectures answer Jackendoff's challenges for cognitive neuroscience. *Joint International Conference on Cognitive Science*.
- Kanerva, P. (2009). Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation* 1(2):139–159.
- Kleene, S. C. (1952). *Introduction to Metamathematics*. North- Holland. The strong three-valued logic system used as the ground for Sutra's polynomial fuzzy connectives (§1.1-1).
- Badreddine, S., Garcez, A. d., Serafini, L., & Spranger, M. (2022). Logic Tensor Networks. *Artificial Intelligence* 303.
- Hájek, P. (1998). *Metamathematics of Fuzzy Logic*. Trends in Logic vol. 4. Kluwer Academic. The standard reference for t-norm-based fuzzy logics (Gödel, Łukasiewicz, product) cited in §1.1-1 to place Sutra's polynomial connectives.
- Heddes, M., Nunes, I., Vergés, P., Kleyko, D., Abraham, D., Givargis, T., Nicolau, A., & Veidenbaum, A. (2023). Torchhd: An open source python library to support research on hyperdimensional computing and vector symbolic architectures. *Journal of Machine Learning Research* 24(255):1–10.
- Li, Z., Huang, J., & Naik, M. (2023). Scallop: A Language for Neurosymbolic Programming. *Proceedings of the ACM on Programming Languages* 7(PLDI):1463–1487. arXiv:2304.04812.
- Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., & De Raedt, L. (2018). DeepProbLog: Neural Probabilistic Logic Programming. *NeurIPS*.
- Serafini, L. & Garcez, A. d. (2016). Logic Tensor Networks: Deep Learning and Logical Reasoning from Data and Knowledge. *NeSy Workshop*.
- van Krieken, E., Acar, E., & van Harmelen, F. (2022). Analyzing Differentiable Fuzzy Logic Operators. *Artificial Intelligence* 302:103602. The differentiable-fuzzy-logic survey cited in §1.1-1; analyzes t-norm-derived AND/OR/IMPLIES operators in the neural-symbolic context and is the closest prior literature to Sutra's polynomial approach.
- Vergés, P., Heddes, M., Nunes, I., Givargis, T., & Nicolau, A. (2023). HDCC: A Hyperdimensional Computing compiler for classification on embedded systems and high-performance computing. arXiv:2304.12398.
- Yang, Z., Ishay, A., & Lee, J. (2020). NeurASP: Embracing Neural Networks into Answer Set Programming. *IJCAI*.
- Plate, T. A. (1995). Holographic reduced representations. *IEEE Transactions on Neural Networks* 6(3):623–641.
- Siegelmann, H. T. & Sontag, E. D. (1992). On the computational power of neural nets. *COLT '92*. Establishes that recurrent neural networks with rational weights are Turing-complete; the result Sutra inherits via tail-recursive loops over a fixed-width state vector.
- Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence* 46(1–2):159–216.

Appendix

Appendix A. Notation: extended layout and primitive operations

We work in a fixed-dimensional real vector space $\mathbb{R}^d$ where d is the substrate's embedding dimension (768 for nomic-embed-text, 384 for all-minilm, 1024 for mxbai-embed-large, 320 for ESM-2). Every Sutra value carries the extended layout $[\,\text{semantic} \mid \text{synthetic}\,]$, a d-dimensional semantic block holding the substrate embedding, concatenated with a small fixed-width synthetic block reserving canonical axes for primitive types (real, imag, truth, char, loop-done) and slot machinery (§3.3). Where notation does not distinguish, "vector" means "the full extended-layout tensor."

The seven primitive operations are:

$$
\begin{aligned}
\text{bind}(r, f) &= R_r\, f, \qquad R_r = \text{QR}(\text{seed} = \text{hash}(r)).Q \\
\text{unbind}(r, v) &= R_r^\top v \\
\text{bundle}(x, y) &= \frac{x + y}{\|x + y\| + \varepsilon} \\
\text{similarity}(x, y) &= \frac{x \cdot y}{\|x\|\,\|y\| + \varepsilon} \\
\text{normalize}(v) &= \frac{v}{\|v\| + \varepsilon}
\end{aligned}
$$

plus the Lagrange Kleene gates (scalar $\to$ scalar, exact on the $\{-1, 0, +1\}^2$ grid, §1.1-1) and the soft-halt cell (state, halt $\to$ state$'$, halt$'$, §3.4).

The Lagrange gates in closed form:

$$
\begin{aligned}
\text{AND}(a, b) &= \tfrac{1}{2}\big(a + b + ab - a^2 - b^2 + a^2 b^2\big) \\
\text{OR}(a, b) &= \tfrac{1}{2}\big(a + b - ab + a^2 + b^2 - a^2 b^2\big) \\
\text{NOT}(a) &= -a \\
\text{XOR}(a, b) &= -ab \\
\text{XNOR}(a, b) &= ab
\end{aligned}
$$

The soft-halt cell update is, in compact form,

$$
\begin{aligned}
s_{t+1} &= R\, s_t && \text{(rotation step)} \\
h_t &= \text{Heaviside}(\text{cond}(s_t)) && \text{(per-tick halt signal)} \\
H_t &= \text{sat}_{[0,1]}\Big(\textstyle\sum_{k \le t} h_k\Big) && \text{(cumulative monotone halt)} \\
\hat{s}_{t+1} &= H_t\, s_t + (1 - H_t)\, s_{t+1} && \text{(soft-mux freeze)}
\end{aligned}
$$

Every right-hand side is a tensor expression with no Python control flow. The compile-time primitives `RotationFor` and `embed` produce constants R_r and basis vectors at compile time and are not part of the runtime tensor graph.

Appendix B. Extended-state-vector layout: per-axis assignments

§3.3 describes the `[semantic | synthetic]` layout in prose. The diagram and per-axis purpose table below give the concrete allocation referenced in `codegen_pytorch.py`:

```
          +-------------------------+----+----+----+----+----+----------+
   value  | semantic block          | R  | I  | T  | C  | L  | slots... |
          +-------------------------+----+----+----+----+----+----------+
```

```
|<-- semantic_dim ------->|<--- synthetic_dim ----------------|>
                            0    1    2    3    4    5..
                          REAL IMAG TRUTH CHAR LOOP_DONE
                                            _FLAG
```

Index	Purpose
`synthetic[0]`	`AXIS_REAL` (real component for int/float/complex)
`synthetic[1]`	`AXIS_IMAG` (imaginary component for complex)
`synthetic[2]`	`AXIS_TRUTH` (fuzzy truth scalar; bool/comparisons)
`synthetic[3]`	`AXIS_CHAR_FLAG` (marks char primitives)
`synthetic[4]`	`AXIS_LOOP_DONE` (substrate-side completion flag)
`synthetic[5..]`	`SLOT_BASE`: disjoint 2D Givens slots for variable storage

At `semantic_dim = 768` (nomic-embed-text), `synthetic_dim = 100` accommodates the five canonical axes plus 47 disjoint Givens slots.

Appendix C. Capacity: full per-substrate sweeps

Cross-substrate decode accuracy at full bundle widths k $\in$ {2, 4, 8, 16, 24, 32, 48}. The four substrates use 84-entry vocabularies (LLM substrates: 84-word noun set spanning animals, foods, objects, places, abstract nouns; ESM-2: 84-sequence amino-acid set covering canonical signal peptides, cell-penetrating peptides, antimicrobial peptides, classic affinity-tag motifs, and deterministic random k-mers). All embeddings are unit-normalized; nomic-embed-text and ESM-2 are additionally mean-centered.

nomic-embed-text (768-d, mean-centered):

k	rotation accuracy	rotation signal cos	Hadamard accuracy	Hadamard signal cos
2	100.0%	+0.703	95.0%	+0.488
4	100.0%	+0.497	95.0%	+0.400
8	100.0%	+0.354	87.5%	+0.307
16	100.0%	+0.251	84.4%	+0.230
24	100.0%	+0.203	60.8%	+0.189
32	99.1%	+0.176	63.1%	+0.167
48	93.3%	+0.144	48.3%	+0.136

all-minilm (384-d):

k	rotation accuracy	rotation signal cos	Hadamard accuracy	Hadamard signal cos
2	100.0%	+0.711	45.0%	+0.386
4	100.0%	+0.506	10.0%	+0.335
8	100.0%	+0.356	7.5%	+0.315
16	92.5%	+0.252	3.1%	+0.299
24	76.2%	+0.203	2.9%	+0.300
32	66.9%	+0.179	2.5%	+0.297
48	42.3%	+0.144	1.7%	+0.294

mxbai-embed-large (1024-d):

k	rotation accuracy	rotation signal cos	Hadamard accuracy	Hadamard signal cos
2	100.0%	+0.708	15.0%	+0.311
4	100.0%	+0.500	2.5%	+0.304
8	100.0%	+0.353	2.5%	+0.295
16	98.8%	+0.251	1.2%	+0.294
24	95.8%	+0.203	0.8%	+0.293
32	85.3%	+0.176	0.9%	+0.292
48	72.1%	+0.146	1.0%	+0.291

ESM-2 small protein language model (320-d, mean-centered):

k	rotation accuracy	rotation signal cos	Hadamard accuracy	Hadamard signal cos
2	100.0%	+0.713	75.0%	+0.470
4	100.0%	+0.501	50.0%	+0.323
8	100.0%	+0.349	28.7%	+0.257
16	90.6%	+0.252	16.2%	+0.185
24	77.1%	+0.205	11.2%	+0.171
32	61.9%	+0.174	6.2%	+0.141
48	44.2%	+0.143	4.2%	+0.117

The signal cosine for Hadamard is comparable to rotation's, but the noise floor is much higher because the elementwise product of correlated real-valued embeddings produces a result that overlaps with many distractors in the codebook rather than near-orthogonally with one.

Appendix D. Crosstalk depth: full per-substrate L-sweep

The §3.2.1 protocol: chain length L $\in$ {1, 2, 4, 8, 16, 32}, 20 trials, bundle width 4 (3 distractors per cycle). Forward-bind through L role rotations bundling 3 distractor (role, filler) pairs at each step; unbind in reverse and decode. Two flavors: *raw* (no cleanup) and *snap* (argmax-cosine cleanup against the codebook after each unbind step).

substrate	L=1 raw	L=2 raw	L=4 raw	L=1 snap	L=2 snap	L=4 snap
nomic-embed-text	100%	100%	20%	100%	10%	0%
all-minilm	100%	100%	5%	100%	0%	0%
mxbai-embed-large	100%	100%	5%	100%	0%	0%

By chain length 8 raw accuracy is at chance (1/84) on all three substrates. Snap is *worse* than raw past chain length 1: a hard codebook commitment converts soft noise into a high-confidence wrong answer that the next unbind cannot recover from. The runtime does not implicitly snap between operations; cleanup is an explicit step the program schedules where it knows the codebook is the right reference. Reproduction script: `experiments/crosstalk_chain.py`; raw JSON in `experiments/crosstalk_chain_results.json`.

Appendix E. Codebook implementation details

The §3.5 codebook is implemented as an embedded vector database (internally SutraDB) shipped as part of the compiler, analogous to SQLite being embedded in an application rather than run as a separate service. The data model is RDF triples with f32-vector literals as the object position, indexed by a built-in HNSW index for nearest-neighbor decode. The on-disk format is a `.sdb` file that travels alongside the compiled Python module; no external service, no separate install, no network dependency. Every embedded string in a Sutra program is inserted with the embedding as the object of a triple typed `<http://sutra.dev/f32vec>`. Strings declared but unused in expressions

are still inserted, so they remain decodable. The compiled module's Python data section never carries the embeddings, they live in the `.sdb` file, an artifact of compilation, not a service the runtime contacts.

`nearest_string` runs over an HNSW (Hierarchical Navigable Small World) approximate-nearest-neighbor graph maintained by the triplestore. HNSW (Malkov & Yashunin, TPAMI 2020) has **O(log N) expected and worst-case query time** under standard graph-construction parameters; it has displaced linear scan as the default ANN index in Faiss, Milvus, Weaviate, Qdrant, and most production vector databases. A 100-string codebook and a 100,000-string codebook have comparable decode latency at runtime, modulo HNSW's tunable `M` (graph degree) and `ef_search` (beam width); the cost difference is roughly one extra graph hop per $10\times$ growth in N.

Appendix F. Worked lowering of a two-field bundled record

The body §4.2 sketches the lowering of $\text{encode2}(r_a, f_a, r_b, f_b) := \text{bundle}(\text{bind}(r_a, f_a), \text{bind}(r_b, f_b))$. Here we trace each stage with the explicit residual.

Stage 1: AST after parse. A tree of `Call` nodes over named identifiers: `Call("bundle", Call("bind", r_a, f_a), Call("bind", r_b, f_b))`.

Stage 2: beta reduction by stdlib inlining. `bind`, `bundle`, and `normalize` are stdlib functions: $\text{bind}(r, f) \equiv \text{RotationFor}(r)\, f$, $\text{bundle}(x, y) \equiv \text{normalize}(x + y)$, $\text{normalize}(v) \equiv v/(\|v\| + \varepsilon)$. After substitution the body becomes

$$\text{normalize}\big(\text{RotationFor}(r_a)\, f_a \;+\; \text{RotationFor}(r_b)\, f_b\big).$$

No `bind` or `bundle` symbol remains; the residual is straight- line algebra over four tensor primitives.

Stage 3: compile-time constant resolution. $\text{RotationFor}(r)$ is a compile-time function returning $R = \text{QR}(\text{seed} = \text{hash}(r)).Q$. The compiler evaluates it for each role at compile time, freezes the results as constant tensors R_a and R_b, and stores them in the rotation cache. The body becomes $\text{normalize}(R_a\, f_a + R_b\, f_b)$, R_a and R_b are now load-bearing constants in the same sense as the weight matrices of a feed-forward network.

Stage 4: peephole fusion. The simplifier recognizes $\text{normalize}\big(\sum_i R_i\, f_i\big)$ as the bundle-of-binds pattern and rewrites it to `_VSA.bundle_of_binds([(R_a, f_a), (R_b, f_b)])`, one kernel launch instead of two matmuls + add + norm.

Stage 5: leaf tensor ops at runtime. `bundle_of_binds` stacks rotations into a (k, d, d) tensor, stacks fillers into (k, d), runs one batched einsum + sum + L2-normalize:

$$\begin{aligned} v \;&=\; \sum_k R_k\, f_k \;=\; \texttt{einsum}(\text{"}\texttt{kij,kj->i}\text{"},\ \text{stack}([\texttt{R}_\texttt{a}, \texttt{R}_\texttt{b}]),\ \text{stack}([\texttt{f}_\texttt{a}, \texttt{f}_\texttt{b}])) \\ \text{encode2} \;&=\; v \,/\, (\|v\| + \varepsilon) \end{aligned}$$

The compiled forward pass for `encode2` is exactly those three torch calls (einsum, linalg.norm, divide) over precomputed R_a, R_b and runtime-supplied f_a, f_b.

Appendix G. §3.6 differentiable-training vocabulary

Twenty categories of fifty words each (992 unique after deduplication), embedded via nomic-embed-text:

- **animal**: dog, cat, bird, fish, horse, lion, tiger, elephant, rabbit, monkey, bear, wolf, fox, deer, mouse, snake, frog, turtle, dolphin, whale, shark, eagle, owl, sparrow, crow, robin, parrot, swan, duck, goose, chicken, cow, pig, sheep, goat, donkey, camel, giraffe, kangaroo, koala, panda, leopard, cheetah, hippopotamus, rhinoceros, antelope, buffalo, hedgehog, squirrel, raccoon
- **vehicle**: car, truck, airplane, boat, bicycle, motorcycle, bus, train, ship, helicopter, tractor, scooter, van, taxi, jeep, sailboat, kayak, canoe, raft, submarine, glider, jet, rocket, spaceship, sled, skateboard, wagon, carriage, chariot, ambulance, firetruck, limousine, minivan,

hatchback, sedan, coupe, convertible, pickup, trailer, ferry, yacht, dinghy, blimp, balloon, hovercraft, tram, moped, tricycle, rollerblade, unicycle

- **food**: apple, bread, cheese, rice, pasta, banana, salad, soup, meat, pizza, sandwich, burger, taco, sushi, cake, cookie, pie, donut, muffin, pancake, waffle, bagel, croissant, omelet, salmon, tuna, beef, pork, lamb, bacon, ham, sausage, steak, lobster, shrimp, crab, oyster, clam, broccoli, carrot, lettuce, tomato, potato, cucumber, onion, garlic, pepper, eggplant, spinach, mushroom
- **color**: red, blue, green, yellow, orange, purple, black, white, brown, pink, gray, cyan, magenta, violet, indigo, turquoise, teal, lavender, maroon, crimson, scarlet, ruby, gold, silver, bronze, copper, beige, tan, ivory, charcoal, navy, sapphire, emerald, jade, olive, lime, mint, coral, peach, plum, mauve, fuchsia, amber, ochre, sienna, mahogany, chocolate, caramel, mustard, azure
- **clothing**: shirt, pants, dress, hat, shoes, jacket, socks, gloves, scarf, belt, sweater, hoodie, jeans, shorts, skirt, blouse, coat, cap, beanie, mittens, tights, leggings, vest, blazer, suit, tuxedo, gown, robe, kimono, kilt, poncho, cloak, cape, sneakers, boots, sandals, slippers, heels, loafers, tie, bowtie, cufflinks, watch, ring, necklace, earrings, bracelet, anklet, brooch, headband
- **weather**: rain, snow, wind, cloud, storm, fog, frost, hail, thunder, lightning, drizzle, downpour, blizzard, hurricane, tornado, cyclone, typhoon, sleet, mist, haze, smog, sunshine, sunlight, sunset, sunrise, dawn, dusk, twilight, breeze, gust, gale, humidity, drought, flood, monsoon, snowfall, snowstorm, rainstorm, sandstorm, heatwave, chill, dew, hailstorm, thaw, overcast, sunny, cloudy, rainy, snowy, windy
- **emotion**: joy, sadness, anger, fear, love, hope, surprise, disgust, pride, envy, happiness, grief, rage, anxiety, affection, despair, delight, shame, guilt, confidence, contentment, jealousy, regret, sorrow, frustration, satisfaction, awe, wonder, gratitude, compassion, sympathy, empathy, irritation, boredom, excitement, enthusiasm, calm, serenity, melancholy, nostalgia, longing, embarrassment, humiliation, indifference, ecstasy, bliss, dread, terror, amusement, loneliness
- **tool**: hammer, saw, drill, wrench, screwdriver, knife, scissors, pliers, axe, shovel, rake, hoe, spade, pickaxe, crowbar, mallet, chisel, sander, level, ruler, vise, clamp, ratchet, socket, awl, scraper, trowel, broom, mop, sponge, bucket, ladder, jackhammer, sledgehammer, paintbrush, roller, stapler, tongs, tweezers, calipers, magnifier, flashlight, multimeter, wirecutter, hacksaw, router, torch, soldering_iron, drillbit, screwbit
- **instrument**: guitar, piano, drum, violin, flute, trumpet, saxophone, harp, cello, clarinet, banjo, mandolin, ukulele, harmonica, accordion, organ, keyboard, synthesizer, xylophone, tambourine, maracas, bongos, marimba, vibraphone, glockenspiel, bagpipes, oboe, bassoon, trombone, tuba, lute, sitar, koto, zither, dulcimer, cymbal, gong, triangle, cowbell, snare, kettledrum, recorder, piccolo, fife, didgeridoo, theremin, viola, double_bass, fiddle, ocarina
- **profession**: doctor, teacher, lawyer, engineer, nurse, chef, artist, scientist, farmer, plumber, electrician, carpenter, mechanic, pilot, sailor, soldier, judge, journalist, writer, poet, painter, sculptor, musician, actor, dancer, singer, photographer, architect, dentist, surgeon, pharmacist, veterinarian, librarian, accountant, banker, broker, programmer, designer, manager, secretary, butcher, baker, gardener, tailor, jeweler, barber, chemist, biologist, physicist, mathematician
- **body_part**: head, hand, foot, eye, ear, nose, mouth, leg, arm, finger, toe, knee, elbow, shoulder, hip, neck, back, chest, stomach, heart, brain, lung, liver, kidney, bone, muscle, skin, hair, throat, jaw, chin, cheek, forehead, eyebrow, eyelash, lip, tongue, palm, wrist, ankle, thumb, heel, spine, rib, scalp, nostril, gum, knuckle, tendon, vein
- **plant**: tree, flower, grass, bush, vine, fern, moss, herb, weed, leaf, stem, branch, bark, blossom, petal, oak, maple, willow, birch, cedar, bamboo, cactus, rose, tulip, daisy, lily, sunflower, orchid, ivy, basil, rosemary, thyme, sage, lavender, dandelion, clover, lotus, magnolia, sycamore, redwood, baobab, eucalyptus, juniper, hemlock, fir, spruce, ash, elm, poplar, chestnut
- **furniture**: chair, table, sofa, bed, desk, shelf, drawer, cabinet, wardrobe, dresser, nightstand, ottoman, bench, stool, recliner, futon, couch, armchair, bookcase, sideboard, buffet, cupboard, hutch, vanity, headboard, footboard, mattress, pillow, cushion, blanket, quilt, comforter, lamp, mirror, rug, carpet, curtain, blind, shutter, hammock, cradle, crib, bassinet, highchair, rocker, loveseat, settee, divan, chaise, headrest

- **building**: house, apartment, mansion, cottage, cabin, hut, igloo, tent, palace, castle, fortress, tower, skyscraper, office, factory, warehouse, store, mall, restaurant, hotel, motel, hospital, school, university, library, museum, theater, stadium, arena, church, temple, mosque, synagogue, cathedral, chapel, monastery, abbey, barn, shed, garage, basement, attic, cellar, lobby, lounge, hallway, corridor, atrium, foyer, balcony
- **country**: France, Germany, Italy, Spain, Portugal, England, Scotland, Ireland, Norway, Sweden, Finland, Denmark, Iceland, Russia, Poland, Greece, Turkey, Egypt, Morocco, Algeria, Kenya, Nigeria, Ethiopia, Ghana, Senegal, Mali, Sudan, Uganda, Tanzania, Madagascar, China, Japan, Korea, Vietnam, Thailand, Malaysia, Indonesia, India, Pakistan, Bangladesh, Iran, Iraq, Israel, Lebanon, Australia, Canada, Mexico, Brazil, Argentina, Chile
- **sport**: football, basketball, baseball, soccer, tennis, golf, hockey, rugby, cricket, volleyball, swimming, running, cycling, skiing, snowboarding, surfing, sailing, rowing, kayaking, climbing, hiking, boxing, wrestling, fencing, archery, shooting, fishing, hunting, polo, badminton, ping_pong, squash, racquetball, lacrosse, handball, dodgeball, kickball, gymnastics, diving, weightlifting, judo, karate, taekwondo, sumo, marathon, triathlon, decathlon, biathlon, skating, bowling
- **drink**: water, juice, milk, tea, coffee, soda, beer, wine, whiskey, vodka, rum, gin, tequila, brandy, cognac, champagne, cocktail, smoothie, milkshake, lemonade, cider, ale, lager, stout, bourbon, scotch, sake, mead, punch, eggnog, kombucha, kefir, espresso, latte, cappuccino, mocha, americano, macchiato, frappe, hot_chocolate, cordial, shake, slushie, syrup, fizz, brew, tonic, infusion, ginger_ale, root_beer
- **metal**: gold, silver, copper, iron, steel, aluminum, brass, bronze, tin, lead, zinc, nickel, platinum, titanium, chromium, mercury, magnesium, lithium, sodium, potassium, calcium, uranium, plutonium, palladium, tungsten, vanadium, cobalt, manganese, beryllium, gallium, indium, antimony, bismuth, cadmium, cerium, neodymium, osmium, rhodium, ruthenium, tantalum, thallium, thorium, yttrium, scandium, hafnium, niobium, molybdenum, rhenium, iridium, rubidium
- **shape**: circle, square, triangle, rectangle, oval, ellipse, pentagon, hexagon, octagon, diamond, rhombus, trapezoid, parallelogram, polygon, sphere, cube, cylinder, cone, pyramid, prism, cuboid, tetrahedron, dodecahedron, icosahedron, octahedron, torus, helix, spiral, crescent, star, heart, arrow, cross, line, curve, arc, ring, loop, knot, dot, vertex, edge, angle, parabola, hyperbola, sine, wave, zigzag, scallop, annulus
- **fabric**: cotton, wool, silk, linen, polyester, nylon, denim, leather, suede, velvet, satin, lace, tweed, cashmere, mohair, fleece, fur, canvas, burlap, jute, flannel, chiffon, organza, taffeta, brocade, damask, paisley, gingham, plaid, herringbone, corduroy, microfiber, spandex, lycra, rayon, viscose, acrylic, polypropylene, jersey, knit, sherpa, gabardine, twill, muslin, gauze, mesh, vinyl, tulle, georgette, voile

Appendix H. Reproduction details and hyperparameters

Per-experiment configuration. All scripts live under `experiments/` in the source repository; each writes a JSON results file to the same directory on completion. RNG seeds are fixed in the source files cited; re-running reproduces the numbers reported in the body to the precision reported.

Experiment	§	Script	Trials / k	Embedding	Optimizer	Seed
Rotation vs Hadamard, LLM	3.2	`rotation_binding_capacity_llm.py`	10 / k	`--embed-text`, all-minilm, mxbai-embed-large		per-script
Rotation vs Hadamard, ESM-2	3.2	`rotation_binding_capacity_esm2.py`	10 / k	facebook/esm2_t6_8M_UR50D		29, 2718
Crosstalk depth	3.2.1	`crosstalk_chain.py`	20 / L	three LLM substrates		per-script

Experiment	§	Script	Trials / k	Embedding	Optimizer	Seed
Differentiable training	3.6	`differentiable_training.py`	1 run × 300 epochs	nomic-embed-text (frozen)	Adam, lr=0.005	42

The differentiable-training run loads twenty learnable prototype vectors (initialized via `torch.randn` × 0.1) and minimizes full-batch cross-entropy over the 992-word vocabulary of Appendix G. Vocabulary embeddings are precomputed once and cached to `.diff_train_embeddings.pt` (3.3 MB) so subsequent runs skip the embed step. Output: weights → `differentiable_training_weights.pt` (3.3 MB), per-epoch metrics → `differentiable_training_results.json`.

Hardware used for the numbers in the body: CPU torch on a single laptop (no CUDA). The full §3.6 run completes in ~3 min wall-clock; the §3.2 capacity sweeps complete in ~2 min per substrate; the §3.2.1 crosstalk sweep completes in ~5 min. Re-running on CUDA should reproduce the same accuracy numbers since the operations are deterministic given a seed.

Appendix I. Demonstration corpus

The smoke test (`examples/_smoke_test.py`) compiles and runs ten `.su` programs end-to-end and asserts each output against a hardcoded expected value. The programs collectively exercise the language features the body claims, with no Python control flow on the runtime path:

Program	Feature exercised
`hello_world.su`	embed + retrieve (minimal program)
`fuzzy_branching.su`	weighted-superposition conditional
`role_filler_record.su`	bind / bundle / unbind on a 3-field record (§2.1)
`classifier.su`	cosine-similarity classifier over a small codebook
`analogy.su`	associative pair memory: capital → country recovery via unbind
`knowledge_graph.su`	(subject, relation, object) triple encode + decode
`predicate_lookup.su`	bind-keyed dictionary read
`fuzzy_dispatch.su`	Lagrange-Kleene-gated dispatch among handlers
`nearest_phrase.su`	top-1 phrase retrieval over a `.sdb` codebook
`sequence.su`	foreach reduction over a list

Loop coverage lives in `examples/do_while_adder.su` and the 23-case `tests/test_loop_function_decl.py` suite. The §3.6 differentiable-training experiment uses the same primitive set the smoke-test programs are built from, no Sutra-runtime extensions, just compilation of `.su` source to PyTorch tensor ops.

Appendix J. Compilation pipeline diagram

The five stages of §4 visualized as a vertical flow with the residual artifact at each stage. Stages (1)–(4) run at compile time; the dashed line marks the compile/runtime boundary; stage (5) is the runtime forward pass.

Appendix K. K=3 rule pipeline diagram

The explicit pipeline graph for the §3.6 differentiable-training classifier at K=3, the smallest setting that exhibits the AND / AND-of-NOTs / softmax / cross-entropy shape that scales unchanged to K=20.

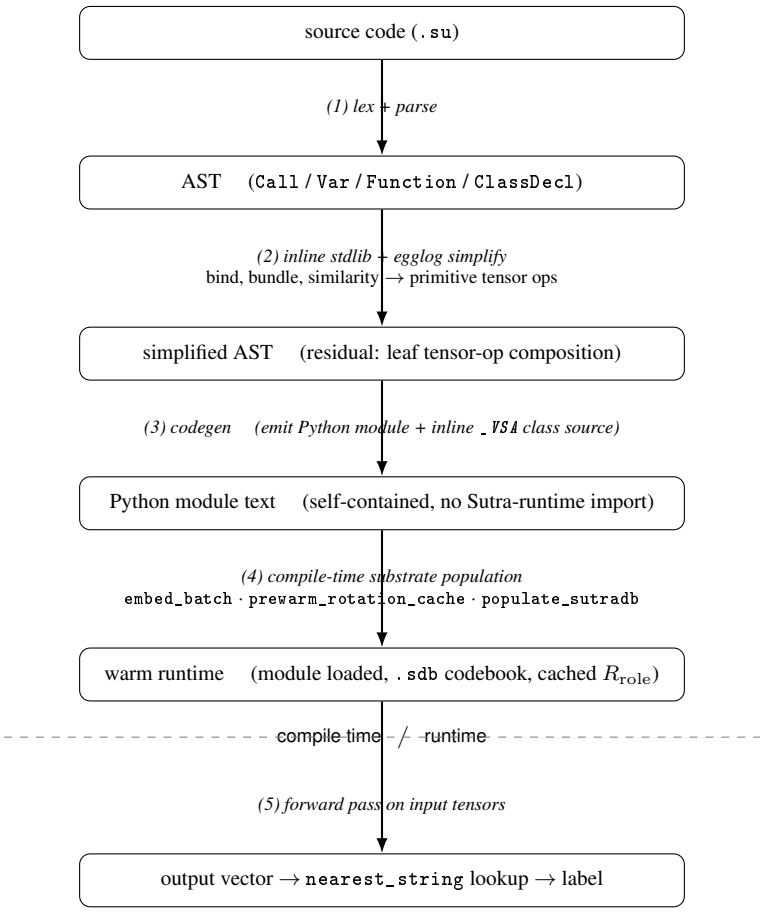

Figure 2: Five-stage compilation pipeline (§4). Boxes are intermediate artifacts; italic labels are the compiler passes that connect them.

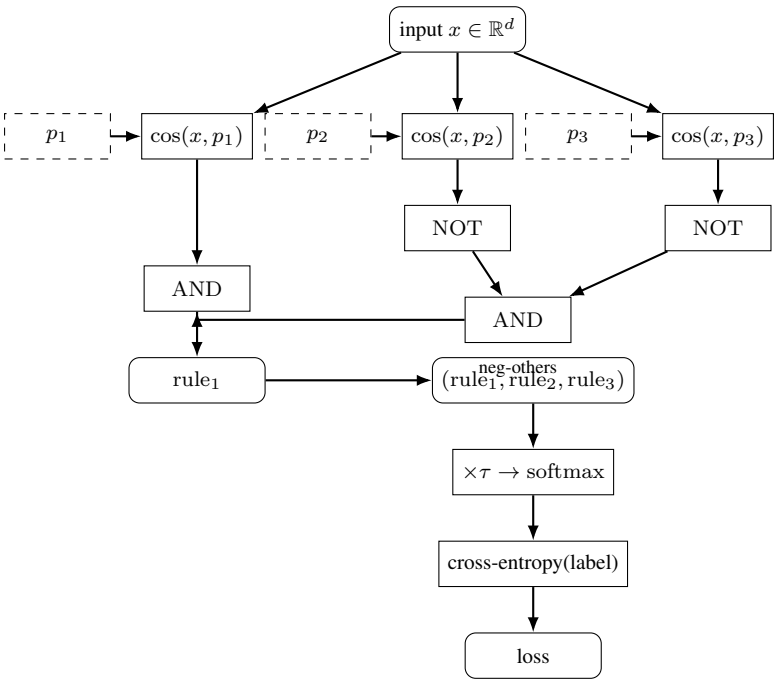

Figure 3: The $K = 3$ rule pipeline. Solid boxes are PyTorch tensor ops; dashed boxes are learnable prototypes. The AND in the leftmost branch combines $\cos(x, p_1)$ with the AND-of-NOTs over the other classes; rule_2 and rule_3 (omitted for clarity) have the symmetric shape. Every edge is a tensor; backprop reaches each p_i through this graph.